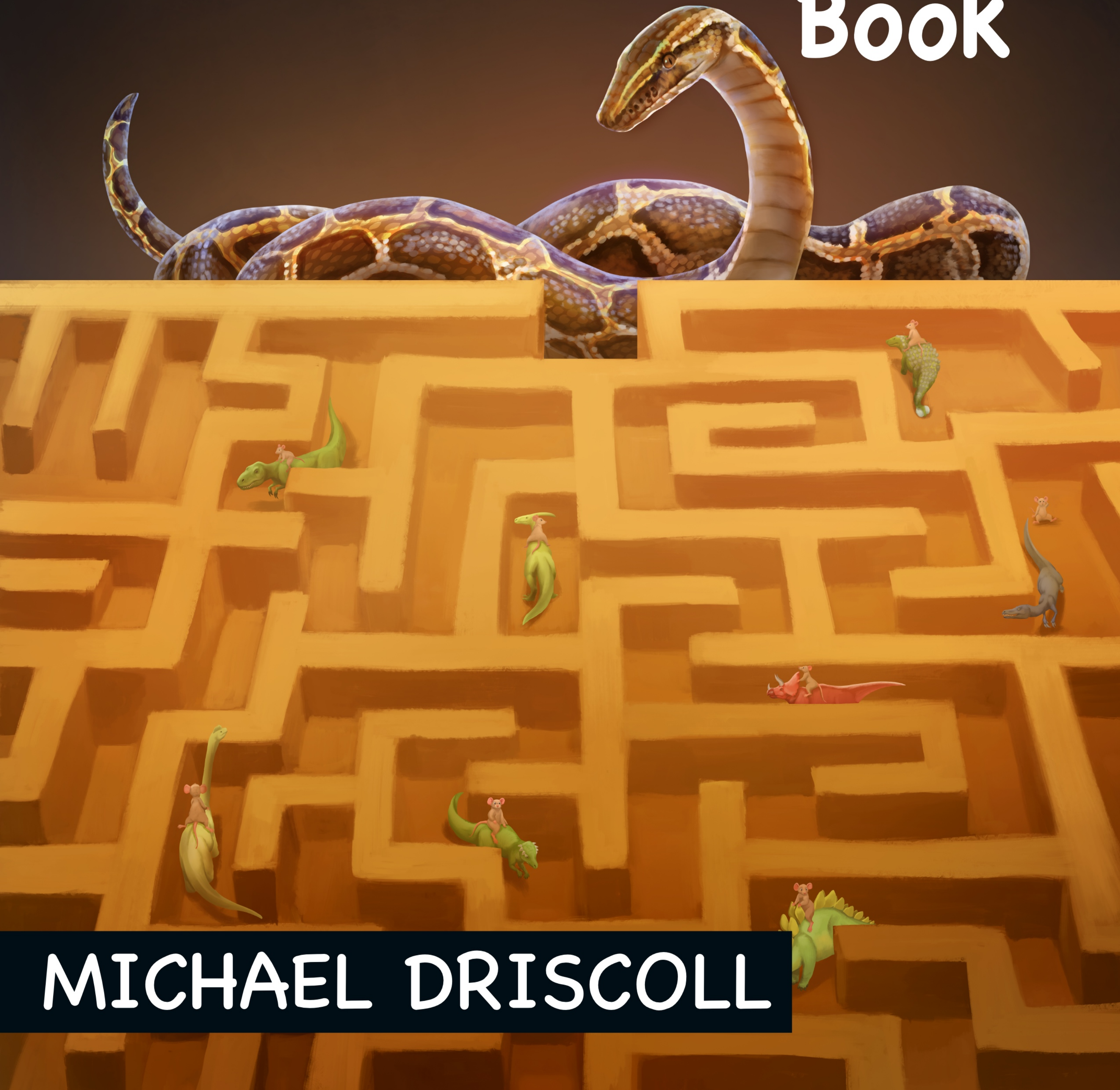
The Python Quiz Book



MICHAEL DRISCOLL

The Python Quiz Book

Michael Driscoll

This book is for sale at http://leanpub.com/pyquiz

This version was published on 2023-02-15

Contents

About the Technical Reviewer

Ethan Furman

Ethan, a largely self-taught programmer, discovered Python around the turn of the century but couldn't explore it for nearly a decade. When he finally did, he fell in love with its simple syntax, lack of boilerplate, and ease with which one can express one's ideas in code. After writing a dbf library to aid in switching his company's code over to Python, he authored PEP 409, wrote the Enum implementation for PEP 435, and authored PEP 461. He was invited to be a core developer after PEP 435, which he happily accepted.

He thanks his wife for joining him in life's great adventure, and his daughter for her permission of the endeavor – without them he would be a much poorer man.

Acknowledgments

I spent a long time in 2021 and 2022 attempting to grow my audience on my website[1] and my Twitter account[2]. For the Twitter audience, I noticed some other people who would post Python quizzes and I thought they were fun.

I started doing my own quizzes and they really took off. I would regularly get more than 100,000 views of those quizzes. So I want to thank all my Twitter followers who enjoyed those quizzes and encouraged me to write this book.

I also want to thank Ethan, my editor, for being willing to read this wacky book and editing for me. Also thank you to those of you who backed the book on Kickstarter or bought early copies of the book. I hope you'll all enjoy the final product.

Mike

[1]https://www.blog.pythonlibrary.org/
[2]https://twitter.com/driscollis

Introduction

Welcome to **The Python Quiz Book**! Quizzes and puzzles are a great way to review your understanding of a concept. Quizzes are also a fun and engaging method of learning new topics. This book aims to have fun and learn about the Python programming language.

The quizzes in this book are not necessarily supposed to guide you in proper programming techniques, and some examples are anti-patterns that you wouldn't use in regular code. However, these code examples are a fun way to introduce different topics, learn how Python works, and see some quirks in the language.

Each chapter in this book will introduce you to a concept in Python using a quiz. There may be a hint or two in case you get stuck. The answer to each quiz is in the back of the book.

Many teachers use quizzes as a way to score their students. In this book, you will score yourself. How well do you know Python? Let's find out!

Audience

This book is for the curious. If you enjoy doing crosswords, playing word games, or writing code tests, you will also enjoy doing these Python quizzes.

The quizzes in this book will also help you learn more about the Python programming language and how it works.

About the Author

Mike Driscoll has been programming with the Python language for over a decade. When Mike isn't programming for work, he writes about Python on his blog[3] and contributes to Real Python. As a technical reviewer, he has worked with Packt Publishing and No Starch Press. Mike has also written several books. Mike is also the founder of Teach Me Python[4] where you can learn the Python programming language through his books and courses.

You can see a full listing of Mike's books on his blog[5].

Mike frequently posts on Twitter about Python, writing, and other topics. You can follow him at @driscollis[6].

[3]https://www.blog.pythonlibrary.org/
[4]https://www.teachmepython.com/
[5]https://www.blog.pythonlibrary.org/books/
[6]https://twitter.com/driscollis

Conventions

Most technical books contain certain types of conventions. The primary convention in this book is code blocks that may or may not have syntax highlighting.

Here is an example:

```python
import platform

def get_operating_system():
    return platform.platform()

if __name__ == "__main__":
    print(f"Your code is running on {get_operating_system()}")
```

These examples allow you to copy and paste the code directly from the book. However, if you have any issues with doing that, then you can learn where to download the code examples in the next section.

Book Source Code

If you ever need to grab a copy of one or more of the Python Quizzes in this book, then all you need to do is go to the book's GitHub repository here:

- https://github.com/driscollis/python_quiz_book

Reader Feedback

I welcome feedback about my writing. If you'd like to let me know what you thought of the book, you can send comments to the following address:

- comments@pythonlibrary.org

Errata

I try my best to avoid publishing errors in my writings, but it happens occasionally. If you happen to see a mistake in this book, feel free to let me know by emailing me at the following:

- errata@pythonlibrary.org

Now let's start learning!

Quiz 1 - Exceptionally Crazy

The Python programming language allows you to catch exceptions using the `try / except` construct. But what happens if you nest exception handlers and throw in a `break` statement too?

Your mission in this quiz is to find the answer to that question!

The Quiz

Are you ready to start solving quizzes? Of course, you are!

Here's your first one:

```python
try:
    for i in range(3):
        try:
            1 / 0
        except ZeroDivisionError:
            raise ZeroDivisionError("Error: You divided by zero!")
        finally:
            print("Finally executed")
            break
except ZeroDivisionError:
    print("Outer ZeroDivisionError exception caught")
```

What will be the output if you run this code?

A. Outer ZeroDivisionError exception caught
B. Error: You divided by zero!
C. Error: You divided by zero!
   Finally executed
D. Finally executed
E. None of the above

When you are ready, flip to the answer section of this book and check your answer!

Hint

If you're stuck, try reading about Python's `break` statement. Exception handling in Python does not usually use the `break` statement.

Answer 1 - Exceptionally Crazy

This is the answer to **Quiz 1 - Exceptionally Crazy**.

Your original question was:

What will be the output if you run this code?

```python
try:
    for i in range(3):
        try:
            1 / 0
        except ZeroDivisionError:
            raise ZeroDivisionError("Error: You divided by zero!")
        finally:
            print("Finally executed")
            break
except ZeroDivisionError:
    print("Outer ZeroDivisionError exception caught")
```

A. Outer ZeroDivisionError exception caught
B. Error: You divided by zero!
C. Error: You divided by zero!
   Finally executed
D. Finally executed
E. None of the above

The Answer

D) Finally executed

Explanation

This code causes the `ZeroDivisionError` to be raised inside the nested exception handler because you are dividing by zero. Your exception handler catches this, but inside the exception handler, you raise `ZeroDivisionError` so that the outer exception handler can run and perhaps handle it differently.

Before you get to the outer exception handler, though, the `finally` statement must run. The `finally` statement **ALWAYS** runs! Here you print out "Finally executed" and then call `break`, and `break` causes the newly raised `ZeroDivisionError` to be discarded and the loop to end.

Try running the code through a debugger and see for yourself.

Quiz 2 - Number Explosion

You will sometimes come across examples of code that use one or two asterisks. Depending on how the asterisks are used, they can mean different things to Python.

Check your understanding of what a single asterisk means in the following quiz!

The Quiz

What will be the output if you run this code?

```python
numbers = range(3)
output = {*numbers}
print(output)
```

A. `{range}`
B. `(range)`
C. `[0, 1, 2]`
D. `(0, 1, 2)`
E. `{0, 1, 2}`

Hint

"Unpacking generalizations" is the term to look up if you get stuck.

Answer 2 - Number Explosion

This is the answer to **Quiz 2 - Number Explosion**.

Your original question was:

What will be the output if you run this code?

```python
numbers = range(3)
output = {*numbers}
print(output)
```

A. `{range}`
B. `(range)`
C. `[0, 1, 2]`
D. `(0, 1, 2)`
E. `{0, 1, 2}`

The Answer

E) `{0, 1, 2}`

Explanation

A single asterisk before a Python dictionary or list is known as the unpacking operator. In this example, you tell Python to unpack three integers (0 - 2) into a set.

Here is the example running in a REPL:

```python
>>> numbers = range(3)
>>> output = {*numbers}
>>> print(output)
{0, 1, 2}
>>> print(type(output))
<class 'set'>
```

The output of the code shows that you have created a set!

You can also use a single asterisk to unpack a dictionary's keys:

~~~~~
>>> my_dict = {1: "one", 2: "two", 3: "three"}
>>> print({*my_dict})
{1, 2, 3}
````

If you want to take your knowledge of unpacking further, it can help to see Python f\
unctions use asterisks:

```python
>>> def my_func(*args):
...     print(args)
...
>>> my_func(1)
(1,)
>>> numbers = range(3)
>>> output = {*numbers}
>>> my_func(output)
({0, 1, 2},)
>>> my_func(*output)
(0, 1, 2)
~~~~~

When you see a single asterisk in a function definition, the asterisk means that the function can take an unlimited number of arguments. In the second example, you pass in the set as a single argument, while in the last example, you use a single asterisk to unpack the numbers and pass them in as three separate arguments.

For more information, see PEP 448 – Additional Unpacking Generalizations[7], which has many more examples!

[7]https://peps.python.org/pep-0448/

Quiz 3 - Letter Explosion

You learned about exploding a `range` object in the last quiz. For this quiz, you will extend that knowledge and discover what happens when you explode a string.

But wait! There's more here than meets the eye. What is up with the numbers in the squiggly brackets?

If you're familiar with string methods, you can figure this quiz out pretty quickly!

The Quiz

What will be output if you run this code?

```python
print("{2}, {1}, {0}".format(*"abc"))
```

A. 'a, b, c'
B. 'c, b, a'
C. ''
D. 'b, a, c'

Hint

The Python documentation[8] is your friend this time.

[8]https://docs.python.org/3/library/stdtypes.html

Answer 3 - Letter Explosion

This is the answer to **Quiz 3 - Letter Explosion**.

Your original question was:

What will be the output if you run this code?

```python
print("{2}, {1}, {0}".format(*"abc"))
```

A. ‘a, b, c’
B. ‘c, b, a’
C. ’’
D. ‘b, a, c’

The Answer

B) ‘c, b, a’

Explanation

Python strings have many methods that you can utilize. In this case, you use the format()[9] method. The squiggly numbers are a type of Format String Syntax[10].

In this case, you are telling Python that this string takes **three** parameters. You are also telling Python where those parameters should be inserted or interpolated into the string.

The string, `"{2}, {1}, {0}"`, is saying that the third parameter is inserted first; the second parameter goes in the middle, and the first parameter is inserted last.

To get three parameters passed to this string, you use the single asterisk and unpack a letter-character string into a function.

You can use that function from the previous quiz here to prove what’s happening:

[9] https://docs.python.org/3/library/stdtypes.html#str.format
[10] https://docs.python.org/3/library/string.html#formatstrings

```
>>> def my_func(*args):
...     print(args)
...
>>> my_func(*"abc")
('a', 'b', 'c')
```

Here you use the single asterisk to pass in three parameters: a, b and c.

Quiz 4 - Type Addition

Python has multiple built-in functions you can use to introspect your code. You'll learn more about those functions in a later quiz.

For this quiz, you'll focus on `isinstance()`, a function you can use to check if a variable is a particular type. For example, you might want to check if a variable is a string or an integer.

Good luck!

The Quiz

What will be the output if you run this code?

```python
my_list = [True, 1, "python", 5, False, {}, True]
integers_found = 0
bools_found = 0

for item in my_list:
    if isinstance(item, int):
        integers_found += 1
    elif isinstance(item, bool):
        bools_found += 1

print(f"{integers_found = } {bools_found = }")
```

A. integers_found = 2 bools_found = 3
B. integers_found = 5 bools_found = 2
C. integers_found = 5 bools_found = 0
D. integers_found = 5 bools_found = 5

Hint

Some types inherit from other types.

Answer 4 - Type Addition

This is the answer to **Quiz 4 - Type Addition**.

Your original question was:

What will be the output if you run this code?

```python
my_list = [True, 1, "python", 5, False, {}, True]
integers_found = 0
bools_found = 0

for item in my_list:
    if isinstance(item, int):
        integers_found += 1
    elif isinstance(item, bool):
        bools_found += 1

print(f"{integers_found = } {bools_found = }")
```

A. integers_found = 2 bools_found = 3
B. integers_found = 5 bools_found = 2
C. integers_found = 5 bools_found = 0
D. integers_found = 5 bools_found = 5

The Answer

C) integers_found = 5 bools_found = 0

Explanation

The trick here is that in Python, the Boolean type is a subclass of `int`, so Booleans are integers. The other trick is that you check if the `item` is an `int` data type BEFORE you check if it is `bool`, so ALL the Boolean types get added to the `integers_found` list.

If you look at `my_list`, you will see two integers: 1 and 5. There are also three Boolean values: True, False and True. Two plus three equals five!

Quiz 5 - Positional Arguments

Python functions can take in several different types of arguments. What are the different argument types called?

If you don’t know, you should review the following:

- Positional arguments
- Keyword arguments

This information may help you solve the quiz, or it may confuse you more. Read the next section when you’re ready to take a crack at it!

The Quiz

How do you call a function like this? Or can you?

```python
def positional(name, age, /, a, b, *, key):
    print(name, age, a, b, key)
```

A. positional(‘Mike’, 17, key=’test’)
B. positional(‘Mike’, 17, 2, b=3, key=’test’)
C. positional(‘Mike’, 2, b=3, key=’test’)
D. None of the above

Hint

The code above is valid syntax in Python, starting in Python 3.8. If you get stuck, you might want to look up PEP 570[11], but only do that as a last resort!

[11] https://peps.python.org/pep-0570/

Answer 5 - Positional Arguments

This is the answer to **Quiz 5 - Positional Arguments**.

Your original question was:

How do you call a function like this? Or can you?

```python
def positional(name, age, /, a, b, *, key):
    print(name, age, a, b, key)
```

A. positional('Mike', 17, key='test')
B. positional('Mike', 17, 2, b=3, key='test')
C. positional('Mike', 2, b=3, key='test')
D. None of the above

The Answer

B) positional('Mike', 17, 2, b=3, key='test')

Explanation

The first two parameters, `name` and `age` are positional only. That means you can't pass them in as keyword arguments, which is why you would see a `TypeError` if you tried to pass either of them in as keywords. The arguments, `a` and `b` can be positional or keyword, whereas `key`, is keyword-only.

The forward slash, /, indicates to Python that all arguments before the forward slash as **positional-only arguments**. Anything following the forward slash is positional or keyword arguments up to the asterisk. The asterisk indicates that everything following it is **keyword-only arguments**.

That means that **B** is the correct way to call this function. There are other ways, of course. But they aren't listed here. Experiment with it yourself and see what alternative methods you can come up with to call this function!

Quiz 6 - Truthy or Falsey

Python supports Booleans in much the same way as other programming languages do. In Python's case, it uses the `True` and `False` keywords. But there is also the concept of truthy and falsey. An example of a truthy value in Python is a number greater than zero (also a number less than zero), and an example of a falsey value would be an empty string. When used in a conditional expression or some functions, these values return `True` or `False`.

Why does this matter? Because you'll write code where you need to know if a string is empty or not. You could check if a dictionary is empty or not too.

With that in mind, you are ready to try this chapter's quiz!

The Quiz

What does the following code print out, if anything?

```python
print(sum([
    all([[]]),
    all([]),
    all([[[]]])
]))
```

A. An exception
B. 1
C. 2
D. 3
E. None of the above

Hint

Did you know that a `bool` type in Python is a subclass of `int`?

Answer 6 - Truthy or Falsey

This is the answer to **Quiz 6 - Truthy or Falsey**.

Your original question was:

What does the following code print out, if anything?

```python
print(sum([
    all([[]]),
    all([]),
    all([[[]]])
]))
```

A. An exception
B. 1
C. 2
D. 3
E. None of the above

The Answer

C) 2

Explanation

Python includes many different built-in functions[12]. You can see two of these functions in this quiz:

- `sum()` - takes an iterable, sums up the contents (if it can), and returns the total
- `all()` - takes an iterable and returns `True` if all the elements in the iterable are true and `False` otherwise.

Try running each of those `all()` functions that are inside the `sum()` function individually:

[12]https://docs.python.org/3/library/functions.html

```
>>> all([[]])
False
>>> all([])
True
>>> all([[[]]])
True
```

What's going on here? To figure that out, you will look at each of these statements one at a time:

- `all([[]])` returns `False` because the passed in list has one element, an empty list `[]`. An empty list is falsey!
- `all([])` returns `True` because the iterable is empty. If you passed it an empty string, it would also return `True`!
- `all([[[]]])` returns `True` because the passed single nested list contains something. In this case, it contains an empty list. But because it contains something, it now evaluates to `True`.

In Python, the `bool` data type is a subclass of `int`. That means that `True` maps to one and `False` maps to `0`.

Look at that code again:

```
print(sum([
    all([[]]),
    all([]),
    all([[[]]])
]))
```

You can rewrite this code to use the return values you saw earlier:

```
print(sum([
    False,
    True,
    True
]))
```

Then you can rewrite it again even simpler:

```python
print(sum([
    0,
    1,
    1
]))
```

What does `0 + 1 + 1` evaluate to? The number `2`, of course!

Quiz 7 - List Popping

The Python list data type is much like an array in other programming languages. The list has many different methods associated with it. You can get a full listing of those methods like this:

```python
my_list = []
print(dir(my_list))
```

One of those methods is called `pop()`. You can pass in an index of the item you want to pop out of the list (if you don't, the last item is chosen). What that means is that you are removing an item from the list.

But what happens if you iterate over the list and attempt to remove some items? If you know the answer to that, this quiz will be easy!

The Quiz

What does the `my_list` look like after this code runs?

```python
my_list = list(range(1, 7))
for index, item in enumerate(my_list): my_list.pop(index)
print(my_list)
```

A. []
B. [1, 3, 5]
C. [2, 4, 6]
D. An IndexError is raised

Hint

What does `enumerate()` do?

Answer 7 - List Popping

This is the answer to **Quiz 7 - List Popping**.

Your original question was:

What does the `my_list` look like after this code runs?

```python
my_list = list(range(1, 7))
for index, item in enumerate(my_list): my_list.pop(index)
print(my_list)
```

A. []
B. [1, 3, 5]
C. [2, 4, 6]
D. An IndexError is raised

The Answer

C) [2, 4, 6]

Explanation

This code runs because you are calling `enumerate()`. The enumerate()[13] function will return an `enumerate` object. That means you are no longer iterating over the list; you are iterating over the `enumerate` object itself!

When you iterate over an `enumerate` object, you get an `index` and an `item` returned. These variables allow you to work with the list without worrying about causing problems.

Try running this code that doesn't use `enumerate()`:

[13]https://docs.python.org/3/library/functions.html#enumerate

```
>>> my_list = list(range(1, 7))
for item in my_list:
...     my_list.pop(item)
...
Traceback (most recent call last):
  Python Shell, prompt 2, line 2
builtins.IndexError: pop index out of range
```

You get an `IndexError` immediately. Never modify an iterable that you are actively iterating over, or you can end up with common or weird logic errors.

When you **do** use `enumerate()`, you can edit the list. In the quiz, you tell Python to remove items from the list.

To see what's happening, rewrite your code like this:

```
>>> my_list = list(range(1, 7))
>>> for index, item in enumerate(my_list):
...     print(f"{index=}")
...     my_list.pop(index)
...     print(my_list)
...
index=0
[2, 3, 4, 5, 6]
index=1
[2, 4, 5, 6]
index=2
[2, 4, 6]
```

The first iteration removes index `0`, which in this case, eliminates the number **1** from the list. The second iteration removes the item at index `1`, which is now three because the list has had the first item removed. The last index to remove is **2**, which is the value `5`. At this point, `my_list` only has three things, so the iteration ends, and you end up with `[2, 4, 6]`.

Quiz 8 - Modulo List Comprehension

Python list comprehensions are great. The list comprehension is a way to take a Python `for` loop and make it a one-liner piece of code.

List comprehensions can include conditional statements that act like filters. They can even contain other list comprehensions, making them difficult to read and comprehend.

If you don't know how to make a list comprehension, you should check out the Python documentation[14] before you try this quiz.

The Quiz

When you run this code, what will print out?

```python
print([x for x in range(10) if x % 2])
```

A. []
B. [1, 3, 5, 7, 9]
C. [0, 2, 4, 6, 8]
D. [2, 6, 10, 14, 18]

Hint

If you get stuck, look up the modulo operator.

[14] https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions

Answer 8 - Modulo List Comprehension

This is the answer to **Quiz 8 - Modulo List Comprehension**.

Your original question was:

When you run this code, what will print out?

```python
print([x for x in range(10) if x % 2])
```

A. []
B. [1, 3, 5, 7, 9]
C. [0, 2, 4, 6, 8]
D. [2, 6, 10, 14, 18]

The Answer

B) [1, 3, 5, 7, 9]

Explanation

Why is the list full of odd numbers? The reason is because of the modulo operator[15]. The modulo operator "yields the remainder from the division of the first argument by the second," per the documentation.

To get a better feel for what's going on, take a gander at these examples:

[15] https://docs.python.org/3.3/reference/expressions.html

```
>>> 1 % 2
1
>>> 2 % 2
0
>>> 3 % 2
1
>>> 4 % 2
0
```

When there is a remainder, it returns a one; otherwise, it returns zero. In **Quiz 6**, you learned that zero is a falsey value and non-zero (i.e. one in this case) values are truthy.

So when you use the `if x % 2` filter in your list comprehension, you are saying to keep the values that return one (the odd ones) and ignore the values that return zero.

There you have it! You have effectively filtered out ALL the even numbers in the range 0-9, and the result is **[1, 3, 5, 7, 9]**.

Quiz 9 - Zipping Lists

Python has lots of great built-in functions[16]. In this quiz you will focus on the built-in `zip()` function. The `zip()` function takes in two or more iterables, iterates over all of them at once in parallel, and produces tuples with an item from each iterable.

If you think you understand the `zip()` function well enough, you can take this quiz!

The Quiz

What does the following code print out?

```python
numbers = [1, 2, 3, 4, 5]
letters = ["a", "b", "c"]
print(list(zip(numbers, letters)))
```

A. Some kind of exception is raised
B. [(1, 'a'), (2, 'b'), (3, 'c')]
C. [(1, 'a'), (2, 'b'), (3, 'c'), (4, ''), (5, '')]
D. [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'a'), (5, 'b')]

Hint

You can read more about `zip()` in the Python documentation[17].

[16]https://docs.python.org/3/library/functions.html
[17]https://docs.python.org/3/library/functions.html#zip

Answer 9 - Zipping Lists

This is the answer to **Quiz 9 - Zipping Lists**.

Your original question was:

What does the following code print out?

```
numbers = [1, 2, 3, 4, 5]
letters = ["a", "b", "c"]
print(list(zip(numbers, letters)))
```

A. Some kind of exception is raised
B. [(1, 'a'), (2, 'b'), (3, 'c')]
C. [(1, 'a'), (2, 'b'), (3, 'c'), (4, ''), (5, '')]
D. [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'a'), (5, 'b')]

The Answer

B) [(1, 'a'), (2, 'b'), (3, 'c')]

Explanation

The `zip()` function is useful for combining two lists. One everyday use case is for creating dictionaries.

Here's how:

```
>>> numbers = [1, 2, 3, 4, 5]
>>> letters = ["a", "b", "c"]
>>> dict(zip(numbers, letters))
{1: 'a', 2: 'b', 3: 'c'}
```

But you'll notice that when you have two lists that are not of equal length, `zip()` will only zip them up to the smaller of the two iterables and then truncate the rest. So you don't get anything past the "c" in the `letters` list. Starting in **Python 3.10** a `strict` flag was added: if the iterables are not the same length, a `ValueError` is raised:

```
>>> numbers = [1, 2, 3, 4, 5]
>>> letters = ["a", "b", "c"]
>>> print(list(zip(numbers, letters, strict=True)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: zip() argument 2 is shorter than argument 1
```

To zip uneven lists using the longest one, you need to use Python's `itertools` module instead. Here's an example:

```
>>> from itertools import zip_longest
>>> list(zip_longest(numbers, letters))
[(1, 'a'), (2, 'b'), (3, 'c'), (4, None), (5, None)]
```

Now when `zip_longest()` hits the end of the shortest iterable, it defaults to adding `None` to the rest of the zipped-up tuples. If you prefer, you can set the fill value to something other to `None` like this:

```
>>> from itertools import zip_longest
>>> list(zip_longest(numbers, letters, fillvalue="Python"))
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'Python'), (5, 'Python')]
```

Use this knowledge wisely!

Quiz 10 - Packing Variables

Packing, unpacking, exploding, contracting. What do these words have to do with programming in Python anyway? It all depends on context! In the case of this chapter's quiz, you are learning more about how to use the asterisk character in Python.

You can use the asterisk to unpack or explode an iterable. But what happens when you are already unpacking an iterable and using an asterisk?

If you know, then this quiz will be a breeze!

The Quiz

What's in the `b` variable?

```
a, *b, c = [1, 2, 3, 4, 5]
```

A. 2
B. [2, 3, 4, 5]
C. [2, 3, 4]
D. What is this madness?

Hint

You can refer back to **Quiz 2 - Number Explosion** for ideas if you get stuck.

Answer 10 - Packing Variables

This is the answer to **Quiz 10 - Packing Variables**.

Your original question was:

What's in the `b` variable?

```
a, *b, c = [1, 2, 3, 4, 5]
```

A. 2
B. [2, 3, 4, 5]
C. [2, 3, 4]
D. What is this madness?

The Answer

C) [2, 3, 4]

Explanation

Sometimes you use an asterisk in Python to extract or unpack items from an iterable. In this quiz, you use them to do the opposite. So, instead of unpacking, you are packing the items into a variable.

Usually, when you want to do multiple assignments in Python, you would have the correct number of variables on the left side and the correct number of items to assign on the right side.

Here's an example:

```
>>> a, b, c = [1, 2, 3]
>>> a
1
>>> b
2
>>> c
3
```

But what happens if too many items are on one side or the other?

```
>>> a, b, c = [1, 2, 3, 4]
Traceback (most recent call last):
  Python Shell, prompt 32, line 1
builtins.ValueError: too many values to unpack (expected 3)
```

You get an exception; that's what happens!

To prevent the exception from happening, you have two choices:

- Make sure that both sides have the same number of variables and items
- Use the asterisk

You can use the asterisk on ANY of the variables on the left, and Python will take care of the rest for you:

```
>>> a, *b, c = [1, 2, 3, 4]
>>> a
1
>>> b
[2, 3]
>>> c
4
>>> a, b, *c = [1, 2, 3, 4]
>>> a
1
>>> b
2
>>> c
[3, 4]
>>> *a, b, c = [1, 2, 3, 4]
>>> a
[1, 2]
>>> b
3
>>> c
4
```

Give it a try and see for yourself!

Quiz 11 - Lambdas and Bools

Many programming languages have the concept of the **lambda** function. In Python, lambda creates an anonymous function. Their syntax looks a bit odd, but it's just taking a simple function and turning it into a one-liner.

But can you call a lambda on the same line you created it? What happens if you try to do a mathematical equation with the result and a Boolean?

If you can answer that question, you'll have no problem with this quiz!

The Quiz

What is the output if you run the following code in a Python REPL?

```
>>> (lambda a, b: a * b)(5, 4) - True
???
```

A. SyntaxError
B. 20
C. 19
D. None of the above

Hint

Lambdas are anonymous functions, and you already know something special about Boolean values in Python.

Answer 11 - Lambdas and Bools

This is the answer to **Quiz 11 - Lambdas and Bools**.

Your original question was:

What is the output if you run the following code in a Python REPL?

```
>>> (lambda a, b: a * b)(5, 4) - True
???
```

A. SyntaxError
B. 20
C. 19
D. None of the above

The Answer

C) 19

Explanation

What's happening here? The first step is to understand what the `lambda` is doing.

Here is the code for the `lambda`:

```
(lambda a, b: a * b)
```

When you create a `lambda` you follow that keyword with one or more variables separated by commas. These variables are the `lambda`'s arguments. After the colon, your code tells the `lambda` what to do with those arguments.

You could rewrite the `lambda` above as a regular function like this:

```
def multiplier(a, b):
    return a * b
```

`multiplier()` is a named function versus an anonymous function or `lambda`.

The quiz question put two numbers in parentheses to pass parameters to the `lambda`. In essence, you are passing in the numbers 5 and 4 for the arguments `a` and `b`, respectively.

When you multiply 5 and 4, you end up with 20. However, that answer is **incorrect** because you are ALSO subtracting `True`. Booleans in Python are subclasses of `int` and map to actual integers. `True` maps to 1, so 20 - 1 = 19. There's your answer!

Quiz 12 - Animaniacal

As a kid, **Animaniacs** was one of the funniest cartoons to watch, and it was something to look forward to after school. In keeping with the joy of quizzes, this puzzle includes some characters from that silly show.

Python has several different operators you can use with several different data types. Do you know which ones work with which data types? This quiz will help you learn if you still need to!

The Quiz

What is the output of the following code when it's run?

```python
names = {"Mike", "Pinky", "Brain", "Dot"}
other_names = {"Brain", "Yakko", "Wacko", "Rita"}
print(names & other_names)
```

A. SyntaxError
B. {'Mike', 'Pinky', 'Brain', 'Dot'}
C. {'Yakko'}
D. {'Brain'}

Hint

When you see curly braces in Python, your first thought is probably of the `dict` data type. However, dictionaries are only one of the data types that use them. There is another!

Answer 12 - Animaniacal

This is the answer to **Quiz 12 - Animaniacal**.

Your original question was:

What is the output of the following code when it's run?

```python
names = {"Mike", "Pinky", "Brain", "Dot"}
other_names = {"Brain", "Yakko", "Wacko", "Rita"}
print(names & other_names)
```

A. SyntaxError
B. {'Mike', 'Pinky', 'Brain', 'Dot'}
C. {'Yakko'}
D. {'Brain'}

The Answer

D) {'Brain'}

Explanation

According to the Python documentation[18] *a set object is an unordered collection of distinct hashable objects.* The set object type has many different methods, with `intersection()`, `union()` and `difference()` being the most well-known.

Less known is that you can use Python operators with sets instead of calling those specific functions. In this quiz, the & operator maps to the `intersection()` method.

When you call `intersection()`, it will return only the items that occur in both sets.

Here's an example showing the result of running the code first with `&` and then again with `intersection()`:

[18] https://docs.python.org/3/library/stdtypes.html#set

```
>>> names = {"Mike", "Pinky", "Brain", "Dot"}
... other_names = {"Brain", "Yakko", "Wacko", "Rita"}
...
>>> (names & other_names)
{'Brain'}
>>> names.intersection(other_names)
{'Brain'}
```

See! They are the same!

Quiz 13 - Set Subtraction

Python sets are so much fun that you can try your luck with another quiz about them. This time you will try subtracting one set from another…if you can!

That is the heart of this quiz. Can you use the minus operator on a set? Or will that cause an error?

Give this quiz a try and show off your knowledge!

The Quiz

What is the output of this code?

```python
names = {"Mike", "Pinky", "Brain", "Dot"}
other_names = {"Brain", "Yakko", "Wacko", "Rita"}
print(names - other_names)
```

A. SyntaxError
B. {'Mike', 'Pinky', 'Brain', 'Dot'}
C. {'Pinky', 'Mike', 'Dot'}
D. None of the above

Hint

Yes, this is another way to express a set method. See if you can figure out which one.

Answer 13 - Set Subtraction

This is the answer to **Quiz 13 - Set Subtraction**.

Your original question was:

What is the output of this code?

```
names = {"Mike", "Pinky", "Brain", "Dot"}
other_names = {"Brain", "Yakko", "Wacko", "Rita"}
print(names - other_names)
```

A. SyntaxError
B. {'Mike', 'Pinky', 'Brain', 'Dot'}
C. {'Pinky', 'Mike', 'Dot'}
D. None of the above

The Answer

C) {'Pinky', 'Mike', 'Dot'}

Explanation

The minus sign DOES work with Python sets. In this case, the minus sign maps to the set's `difference()` method.

You can see that in action by taking a gander at the following REPL session:

```
>>> names = {"Mike", "Pinky", "Brain", "Dot"}
>>> other_names = {"Brain", "Yakko", "Wacko", "Rita"}
>>> names - other_names
{'Pinky', 'Mike', 'Dot'}
>>> names.difference(other_names)
{'Pinky', 'Mike', 'Dot'}
```

Here you use the minus operator on the two sets. Then you use the `difference()` method on the `names` set. The result is the same in both cases.

Quiz 14 - Nesting Ternaries

Conditional expressions in Python are the primary way to have your software programs take different courses of action.

Here is an example:

```
>>> temp = 100
>>> if temp >= 212:
...     print("You hit the boiling point")
... elif 32 < temp < 212:
...     print("Water is still liquid")
... else:
...     print("Water is frozen")
...
Water is still liquid
```

Starting in Python 3.10, you can use structural pattern matching[19] instead of conditional statements if you want to.

But that's not the point of this quiz. Instead, this quiz aims to teach you what a **ternary** is. A ternary is a fancy word for a one-line conditional expression.

Can they be nested? If so, do you know how to read them correctly? Try and see!

The Quiz

What is the output when you run this code?

```
A = False
B = True
print("one" if A else "Python" if B else "Ciao!")
```

A. True
B. 'one'
C. 'Python'
D. 'Ciao!'

Hint

There is no hint here other than to read the code carefully.

[19] https://peps.python.org/pep-0622/

Answer 14 - Nesting Ternaries

This is the answer to **Quiz 14 - Nesing Ternaries**.

Your original question was:

What is the output when you run this code?

```python
A = False
B = True
print("one" if A else "Python" if B else "Ciao!")
```

A. True
B. 'one'
C. 'Python'
D. 'Ciao!'

The Answer

C) 'Python'

Explanation

If you had trouble with this quiz, you should know that a quick way to make a ternary easier to read is to turn it into a regular conditional statement.

Here's what it would look like:

```python
>>> A = False
>>> B = True
>>> if A:
...     print("one")
... elif B:
...     print("Python")
... else:
...     print("Ciao!")
...
Python
```

If `A` is `True`, it prints ‘one’; else if `B` is `True`, it prints ‘Python’. If neither is `True`, the code will print ‘Ciao!’

The `B` variable is `True`, so it prints ‘Python’!

Quiz 15 - Type Multiplication

This quiz tests your knowledge about what data types you can multiply. It's not an extensive analysis, but the data types are well-represented.

Which types can you multiply in Python? Prove your knowledge by solving this quiz!

The Quiz

Which of the following raises a `Type Error` in Python?

A. 5 * True
B. 5 * "Spam"
C. 5 * [1, 2]
D. 5 * None
E. C and D

Hint

Use your knowledge of Python. If you don't know, that's okay too.

Answer 15 - Type Multiplication

This is the answer to **Quiz 15 - Type Multiplication**.

Your original question was:

Which of the following raises a `Type Error` in Python?

A. 5 * True
B. 5 * “Spam”
C. 5 * [1, 2]
D. 5 * None
E. C and D

The Answer

D) 5 * None

Explanation

If you didn’t know the answer to this quiz, that’s okay. The best way to find out is to try running each of the statements above:

```
>>> 5 * True
5
```

The `bool` type in Python subclasses `int`. So `True` maps to one. Thus, 5 * 1 = 5. No `Type Error`!

Try multiplying a string:

```
>>> 5 * 'Spam'
'SpamSpamSpamSpamSpam'
```

That worked! As an aside, the name of the Python programming language comes from the original creator’s love of the **Monty Python** show. When you multiply a string, you create a new string with the same characters multiplied multiple times. In this example, “Spam” is multiplied five times with no spaces.

Now try multiplying a list of integers:

```
>>> 5 * [1, 2]
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

If you are familiar with matrix math, you might have thought this would return [5, 10]. Or you have thought it was the nefarious `Type Error`. But it was neither. Instead, this multiplication makes a list five times longer and repeats its elements five times over.

Finally, you get to the correct answer:

```
>>> 5 * None
Traceback (most recent call last):
  Python Shell, prompt 91, line 1
builtins.TypeError: unsupported operand type(s) for *: 'int' and 'NoneType'
```

You cannot multiply an integer and a `NoneType`. It's just not allowed!

Quiz 16 - f-string Formatting

Python works great for parsing strings. Python is also great at string interpolation, a fancy word for inserting small strings into other strings.

Python has three different ways to control string formatting. You can read all about them in Mike Driscoll's other book, Python 101[20]. Suffice it to say the three methods are:

- Using `%s` (ancient)
- Using the string's `format()` method (slightly newer)
- Using f-strings (the newest and most popular)

This quiz focuses on what you can or cannot do with f-strings. Hopefully, you are well-versed in what f-strings let you do!

The Quiz

What will be the output when you run this code?

```
message = "hi"
fill = "s"
align = "<"
width = 10
print(f"{message:{fill}{align}{width}}")
```

A. Syntax Error
B. 'hi'
C. hissssssss
D. hiss
E. None of the above

Hint

PEP 498 - Literal String Interpolation[21] is an excellent place to check for ideas.

[20]https://driscollis.gumroad.com/l/pypy101/
[21]https://peps.python.org/pep-0498/

Answer 16 - f-string Formatting

This is the answer to **Quiz 16 - f-string Formatting**.

Your original question was:

What will be the output when you run this code?

```python
message = "hi"
fill = "s"
align = "<"
width = 10
print(f"{message:{fill}{align}{width}}")
```

A. Syntax Error
B. 'hi'
C. hisssssss
D. hiss
E. None of the above

The Answer

C) hisssssss

Explanation

Python f-strings do allow you to nest formatting inside of curly braces. Typically, you insert a string with no additional formatting applied.

Here's an example:

```python
>>> message = "Hi"
>>> name = "Mike"
>>> print(f"{message} {name}")
Hi Mike
```

In this example, you insert two variables:

- `message`
- `name`

There is no additional formatting done to the string. But if you DID want to, you would put a colon following the variable name you want to format.

What if you wanted to make one of the strings that you are inserting a specific width? You can do that by inserting the colon, followed by the width of the string:

```
>>> message = "Hi"
>>> name = "Mike"
>>> print(f"{message:10} {name}")
Hi         Mike
```

Here the first variable, `message`, is set to be ten characters wide even if the variable itself is not. You didn't provide a fill character, so Python inserts spaces by default.

You can right or left justification using `>` or `<` respectively (or `^` for centering):

```
>>> message = "Hi"
>>> f"{message:<10}"
'Hi        '
>>> f"{message:>10}"
'        Hi'
>>> f"{message:^10}"
'    Hi    '
```

Note that you still need to specify a fill character.

The fill character goes before the justification:

```
>>> f"{message:n<10}"
'Hinnnnnnnn'
>>> f"{message:n>10}"
'nnnnnnnnHi'
>>> f"{message:n^10}"
'nnnnHinnnn'
```

Now, put that all together, and you can insert those values using the nested curly braces syntax you found in the quiz above.

Quiz 17 - f-strings with Dates

Python includes a robust module in its standard library for working with dates called `datetime`. You can take those `datetime` objects and format them in many different ways.

This quiz will test your knowledge of `datetime` format codes. Does the one in the quiz exist? If it does, do you know how that code will affect the output?

Let's find out!

The Quiz

What will this formatted string look like?

```python
# 17_fstring_dates.py

import datetime

day = datetime.datetime(2021, 11, 20)
print(f"{day} was a {day:%A}")
```

A. 11/20/2021 was a Saturday
B. 2021-11-20 00:00:00 was a Saturday
C. 2021-11-20 was a Saturday
D. 20-11-2021 00:00:00 was a Saturday

Hint

The `datetime` module has some format codes[22] that may help you.

[22]https://docs.python.org/3/library/datetime.html#strftime-and-strptime-format-codes

Answer 17 - f-strings with Dates

This is the answer to **Quiz 17 - f-strings with Dates**.

Your original question was:

What will this formatted string look like?

```python
# 17_fstring_dates.py

import datetime

day = datetime.datetime(2021, 11, 20)
print(f"{day} was a {day:%A}")
```

A. 11/20/2021 was a Saturday
B. 2021-11-20 00:00:00 was a Saturday
C. 2021-11-20 was a Saturday
D. 20-11-2021 00:00:00 was a Saturday

The Answer

B) 2021-11-20 00:00:00 was a Saturday

Explanation

To understand what is happening in this code, you need to read up on the `datetime` format codes[23].

The documentation says that `%A` will give the *weekday as locale's full name* (i.e. Monday, Tuesday, Wednesday, etc). You can see that in action by calling a `datetime` object's `strftime()` method using the same format code:

[23]https://docs.python.org/3/library/datetime.html#strftime-and-strptime-format-codes

```
>>> import datetime
>>> day = datetime.datetime(2021, 11, 20)
>>> day.strftime("%A")
'Saturday'
```

You can use those same codes in Python f-strings! To use a `datetime` format code, you need to insert a `datetime` object into a string followed by a colon and then one or more of the `datetime` format codes.

Here are a couple of examples:

```
>>> import datetime
>>> day = datetime.datetime(2021, 11, 20)

>>> f"{day:%B} {day:%d} is a {day:%A}"
'November 20 is a Saturday'

>>> f"{day:%A}"
'Saturday'
```

Try out some of the format codes on your own and see what you can come up with!

Quiz 18 - f-string Rounding

You may not have known it, but number rounding pops up in the craziest places. You'll find rounding when you pump gas, calculate pressure, and in many other places.

This quiz, for example. What kind of rounding does the `round()` function do? It's a built-in function in Python. You don't need to import it to use `round()`.

But it pays to understand what kind of rounding it does!

The Quiz

What is the output of the following code?

```python
number = 10.125
print(f"{round(number):.2f}")
```

A. 10.00
B. 10.13
C. 10.10
D. 10.20

Hint

You should read up on Banker's Rounding if you get stuck with this one.

Answer 18 - f-string Rounding

This is the answer to **Quiz 18 - f-string Rounding**.

Your original question was:

What is the output of the following code?

```python
number = 10.125
print(f"{round(number):.2f}")
```

A. 10.00
B. 10.13
C. 10.10
D. 10.20

The Answer

A) 10.00

Explanation

The first step in understanding Python's `round()` function is by running a few examples through it. Here are a small sampling of examples:

```python
>>> round(1.5)
2
>>> round(1.2)
1
>>> round(2.5)
2
>>> round(2.7)
3
```

Wait a minute! When you round 1.5, it goes up to 2, but when you round 2.5, it also goes to 2! Python doesn't round the 0.5 up in all circumstances. Instead, Python follows **banker's rounding**, which states that when the number ends in **.5**, it will round to the nearest even number.

When you round 1.5, it goes up because 2 is the nearest round number. When you round 2.5, the nearest round number is also 2.

But what happens in this quiz? You're not getting an integer! When you round **10.125**, shouldn't it be **10**?

Let's find out:

```
>>> round(10.125)
10
```

It turns out that you are indeed rounding down to 10. So why are there some decimal places?

That happens because you have turned on floating point formatting in your string with the following: **:.2f**. Using a colon inside curly braces in an f-string tells Python that you want to do special formatting.

In this case, you are telling Python to make the number into a floating point number with two decimal places.

Try re-running the code without using the `round()` function:

```
>>> number = 10.125
>>> print(f"{number:.2f}")
10.12
```

This time there is no rounding. Python truncates the third decimal's place off entirely!

It would be best if you tried doing some formatting yourself using different numbers and different amounts of floating point precision until you feel you completely understand what's happening.

Quiz 19 - Boolean Math

You have seen some silly quizzes involving Python's `bool` type. In this quiz, you'll see another example.

Why are there multiple quizzes on the same topic? The reason is that repetition is a well-proven method for helping students learn a particular topic or skill.

If you remember the previous quizzes, this one will be straightforward. If not, then you may want to go back and re-read a few of them.

The Quiz

What will this Python code print out?

```python
print(True + True + False - True)
```

A. 1
B. 2
C. 0
D. None of the above

Hint

The `bool` data type is subclassed from `int`.

Answer 19 - Boolean Math

This is the answer to **Quiz 19 - Boolean Math**.

Your original question was:

What will this Python code print out?

```python
print(True + True + False - True)
```

A. 1
B. 2
C. 0
D. None of the above

The Answer

A) 1

Explanation

Python's `bool` data type subclasses the `int` data type. That means that `True` and `False` behave as integers in Python.

Here is how that maps out:

- `True` = 1
- `False` = 0

With that knowledge, you can replace the bools in the code above with the following equation:

```
1 + 1 + 0 - 1
```

One plus one is two, plus zero is still two. Take away one, and your total is now one!

Quiz 20 - Dictionary Explosion

That sneaky asterisk operator keeps cropping up. You may remember that you can use an asterisk in Python to unpack or explode an iterable, and you can also use asterisks to make a function accept multiple arguments.

But what happens when you use an asterisk with a dictionary? Is that iterable too? Time to find out!

The Quiz

What happens when you "explode" or "unpack" a Python dictionary?

```python
my_dict = {1: "Charles", 2: "Babbage"}
print(*my_dict)
```

A. `{1: "Charles", 2: "Babbage"}`
B. 1, 2
C. 'Charles', 'Babbage'
D. 1, 'Charles', 2, 'Babbage'

Hint

What happens when you use a Python `for` loop on a dictionary?

Answer 20 - Dictionary Explosion

This is the answer to **Quiz 20 - Dictionary Explosion**.

Your original question was:

What happens when you "explode" or "unpack" a Python dictionary?

```python
my_dict = {1: "Charles", 2: "Babbage"}
print(*my_dict)
```

A. `{1: "Charles", 2: "Babbage"}`
B. 1, 2
C. 'Charles', 'Babbage'
D. 1, 'Charles', 2, 'Babbage'

The Answer

B) 1, 2

Explanation

The easiest way to understand what is happening in this quiz's code is to try iterating over a dictionary.

Here's an example:

```python
>>> my_dict = {1: "Charles", 2: "Babbage"}
>>> for key in my_dict:
...     print(key)
...
1
2
```

When you iterate over a dictionary, you iterate over its keys.

If you wanted to iterate over the values, you would need to be more explicit and call the dictionary's `values()` method:

```
>>> for value in my_dict.values():
...     print(value)
...
Charles
Babbage
```

Alternatively, you might want to iterate over the keys and values, in which case, you'd use the `items()` method:

```
>>> for key, value in my_dict.items():
...     print(f"{key=} {value=}")
...
key=1 value='Charles'
key=2 value='Babbage'
```

But in this quiz, you didn't use ANY dictionary methods whatsoever! Instead, you used an asterisk, which told Python that you wanted to unpack the keys from the dictionary.

Which, in this case, happens to be **1, 2**.

Quiz 21 - Fun with Walruses

Variable assignment is a basic part of coding in Python and all other languages. You create a variable, use an equals sign (AKA the assignment operator), followed by the value you want to assign to the variable.

Python 3.8 released a new syntax feature that allowed the developer to assign values inside an expression. This syntax was called an **Assignment Expression** and it uses the `:=` syntax, dubbed the "walrus operator".

Here is an example usage of an assignment expression:

```
[y := f(x), y**2, y**3]
```

That may help you out with this quiz, or maybe not. You'll have to think about this code a bit and figure out what happens in an assignment expression with multiple values on one side or the other. Good luck!

The Quiz

What is the output of the `print` function?

```
(a := 6, 9)
(a, b := 16, 19)
print(f"{a=} {b=}")
```

A. An exception is raised
B. a=6 b=9
C. a=6 b=19
D. a=6 b=16
E. a=9 b=19

Hint

Check out PEP 572[24] for more information about the weird syntax in this quiz.

[24]https://peps.python.org/pep-0572/

Answer 21 - Fun with Walruses

This is the answer to **Quiz 21 - Fun with Walruses**.

Your original question was:

What is the output of the `print` function?

```
(a := 6, 9)
(a, b := 16, 19)
print(f"{a=} {b=}")
```

A. An exception is raised
B. a=6 b=9
C. a=6 b=19
D. a=6 b=16
E. a=9 b=19

The Answer

D) a=6 b=16

Explanation

If you read through PEP 572[25], which defines assignment expressions, you'll see a section called **Exceptional cases**, which has an example similar to the one in this quiz. They call this syntax "Valid, though not recommended".

That sums up this code well. You would **not** write an assignment expression like the one you see in this code in your production code, and it's terrible form.

It may help you understand how assignment expressions work.

A good way to figure out this quiz is to run it in smaller pieces. You can start by running the first expression in your Python REPL:

[25]https://peps.python.org/pep-0572/

```
>>> (a := 6, 9)
(6, 9)
>>> a
6
```

The REPL tells you that you constructed a tuple (`(6, 9)`, which was immediately discarded), and during the tuple creation the variable `a` was assigned the value `6`.

Now run the second assignment expression in your REPL and inspect the variables:

```
>>> (a, b := 16, 19)
(6, 16, 19)
>>> a
6
>>> b
16
```

Once again we see a tuple was created, this time with the value from `a`, `16`, and `19`. The value `16` was assigned to `b` by the walrus operator, and the `19` was discarded after being displayed.

Now you know what the variables are and why **D** is the correct answer.

Quiz 22 - f-string Justification

Software developers and engineers work with text all the time. They read the text in, parse the data, and reformat it. Some developers parse text all day long in complex ways.

There are even subsets of computer science dedicated to text, such as Natural Language Processing (NLP) and some types of machine learning.

But that's not what this quiz is about. This quiz is about f-strings and that pesky colon. What does that do again? Hopefully, you'll remember from one of the previous quizzes.

The Quiz

What happens when you run this code?

```python
language = "Python"
print(f"{language:*^30}")
```

A. Python^30
B. Python
C. `************Python************`
D. `Python******************************`

Hint

The colon in the string interpolation is your clue!

Answer 22 - f-string Justification

This is the answer to **Quiz 22 - f-string Justification**.

Your original question was:

What happens when you run this code?

```python
language = "Python"
print(f"{language:*^30}")
```

A. Python^30
B. Python
C. `************Python************`
D. `Python******************************`

The Answer

C. `************Python************`

Explanation

Quiz 16 explained what's going on in this quiz quite well, but here's the long and short of it:

- When you see a colon inside of curly braces, it tells you that you'll be doing special formatting
- If a character follows the colon, that is a fill character
- After the fill character comes the justification character
- The width is specified after the justification character

Here are the valid justification characters:

- ^ - center
- < - left justified
- > - right justified

Now let's look at an example that shows all three of the justification characters in action:

```
>>> language = "Python"
>>> print(f"{language:*^30}")
************Python************

>>> print(f"{language:*<30}")
Python************************

>>> print(f"{language:*>30}")
************************Python
```

In this quiz, you created a string consisting of 30-characters. You centered the substring, “Python”, in the middle of the 30 characters. That’s all there is to it!

Quiz 23 - What's Callable?

Python has the concept of objects that are callable and those that are not. You can add parentheses to a callable object, and Python won't throw an exception.

This quiz tests your knowledge of what is callable in Python and what is not.

The Quiz

What is the output of the following code?

```python
objects = [int, "", 1, str]
print([callable(obj) for obj in objects])
```

A. [True, True, True, True]
B. [True, False, False, True]
C. [False, True, False, True]
D. [False, True, True, False]

Hint

Check out Python's built-in functions[26].

[26] https://docs.python.org/3/library/functions.html

Answer 23 - What's Callable?

This is the answer to **Quiz 23 - What's Callable?**.

Your original question was:

What is the output of the following code?

```python
objects = [int, "", 1, str]
print([callable(obj) for obj in objects])
```

A. [True, True, True, True]
B. [True, False, False, True]
C. [False, True, False, True]
D. [False, True, True, False]

The Answer

B) [True, False, False, True]

Explanation

There are two primary callable objects in Python:

- functions
- classes

A string or an integer itself is not callable. But the `int()` and `str()` functions are.

Most variables aren't callable:

```
>>> a = {}
>>> b = "Python"
>>> c = (1, 2, 3)
>>> callable(a)
False
>>> callable(b)
False
>>> callable(c)
False
```

But you can turn a variable into a callable by making it an alias for a function or class.

Here are a couple of examples:

```
>>> my_int = int
>>> callable(my_int)
True
>>> def adder(a, b):
...     return a + b
...
>>> adder_alias = adder
>>> callable(adder_alias)
True
```

Spend a few moments and think about other Python objects that may or may not be callable. Then use the `callable()` function to test your theories!

Quiz 24 - Shadows

This quiz may prove to be your undoing. The code in this quiz demonstrates a problem many Python developers encounter from time to time in their careers. Beginners are the most likely to encounter this sort of error as they are learning the language. However, even experienced developers will occasionally get tripped up by this problem.

If you're experienced, you will spot the issue right away. Is it a logic error? A syntax error? Or something else entirely?

The Quiz

What is the output of the following code?

```python
min = 0
numbers = list(range(5))
print(min(numbers))
```

A. 0
B. TypeError
C. SyntaxError
D. RuntimeError

Hint

Do you know all of Python's built-in functions?

Answer 24 - Shadows

This is the answer to **Quiz 24 - Shadows**.

Your original question was:

What is the output of the following code?

```python
min = 0
numbers = list(range(5))
print(min(numbers))
```

A. 0
B. TypeError
C. SyntaxError
D. RuntimeError

The Answer

B) TypeError

Explanation

This quiz is an example of **shadowing a builtin**. Python has a lot of builtin functions[27]. One of them is the `min()` function.

In this quiz, you create a variable called `min` and set its value to `0`. When you do that assignment, you create an object in the local namespace with the name of `min`. When Python looks for `min` it will find your variable and use that, which means it won't find the `min()` function in the built-ins namespace. It's called "shadowing" because, from that line of code, Python cannot see its built-in function `mi())`. This sort of thing is easier to see in three lines of code. But when you have a file with a few hundred lines of code, it can be more difficult to figure out what is happening.

When you attempt to call `min()` in the last line of this quiz, you will see the following:

```
builtins.TypeError: 'int' object is not callable
```

That is telling you that an integer object is not callable. You can't run an integer, so it throws an exception.

[27]https://docs.python.org/3/library/functions.html

Quiz 25 - Dictionary Keys

Python has several different data types. One of those data types is the Python dictionary! The humble dictionary is a mapping data type that contains a series of keys and values.

You can use dictionaries in many different ways in Python. The dictionary has lots of great methods.

But this quiz is about dictionary basics. How do you add key/value pairs to a dictionary? What happens when you do that?

If you know the answers to those questions, this quiz will be a breeze!

The Quiz

What are the contents of this Python dictionary?

```
languages = {}
languages[1.1] = "C++"
languages[2.0] = "PHP"
languages[2] = "Python"
print(languages)
```

A. `{1.1: 'C++', 2.0: 'PHP', 2: 'Python'}`
B. `{1.1: 'C++', 2.0: 'PHP'}`
C. `{1.1: 'C++', 2: 'Python'}`
D. `{1.1: 'C++', 2.0: 'Python'}`
E. None of the above

Hint

It would help if you had a good understanding of equality and precedence in Python to figure this one out.

Answer 25 - Dictionary Keys

This is the answer to **Quiz 25 - Dictionary Keys**.

Your original question was:

What are the contents of this Python dictionary?

```python
languages = {}
languages[1.1] = "C++"
languages[2.0] = "PHP"
languages[2] = "Python"
print(languages)
```

A. `{1.1: 'C++', 2.0: 'PHP', 2: 'Python'}`
B. `{1.1: 'C++', 2.0: 'PHP'}`
C. `{1.1: 'C++', 2: 'Python'}`
D. `{1.1: 'C++', 2.0: 'Python'}`
E. None of the above

The Answer

D) {1.1: 'C++', 2.0: 'Python'}

Explanation

Do you know what happens in a dictionary when you set its value? What about when you set that key to something else?

Let's find out in the Python REPL:

```
>>> numbers = {1: "one", 2: "two"}
>>> numbers
{1: 'one', 2: 'two'}

>>> numbers[3] = "three"
>>> numbers
{1: 'one', 2: 'two', 3: 'three'}

>>> numbers[3] = "four"
>>> numbers
{1: 'one', 2: 'two', 3: 'four'}
```

Here you create a dictionary with two key/value pairs. Then you add a third key/value pair with the number 3 as the key and “three” as the value.

Finally, you set the key of 3 to “four” which changes the value from “three” to “four”.

What does all this have to do with the quiz, though? It all comes down to how Python interprets numbers!

```
>>> 2 == 2.0
True
>>> 1 == 1.000
True
```

The integer 2 equals the float 2.0. So when you “add” the integer 2 to your dictionary, Python sees that as you want to replace the value of the 2.0 key with the new value of `'Python'`.

Quiz 26 - List Mutation

Did you know that when you change a list's contents, that is called **mutating** the list? Mutation happens when you add or remove items from the list.

Python lists have many fantastic methods to mutate the list and do other operations to the list.

This quiz will test what you know about the humble `append()` list method. Let's see how you will do!

The Quiz

What is the output of this code?

```python
my_list = list(range(1, 7))
my_list = my_list.append(8)
print(my_list)
```

A. [1, 2, 3, 4, 5, 6, 7, 8]
B. [1, 2, 3, 4, 5, 6, 7]
C. None
D. [8]

Hint

What does the `append()` method return?

Answer 26 - List Mutation

This is the answer to **Quiz 26 - List Mutation**.

Your original question was:

What is the output of this code?

```python
my_list = list(range(1, 7))
my_list = my_list.append(8)
print(my_list)
```

A. [1, 2, 3, 4, 5, 6, 7, 8]
B. [1, 2, 3, 4, 5, 6, 7]
C. None
D. [8]

The Answer

C) None

Explanation

What's happening here? Doesn't the `append()` method add an item to the end of the list?

Of course, it does! But while `append()` adds the item to the list in-place, the method itself returns `None`! You are assigning that return value of `None` back to the `my_list` variable and replacing your Python list.

You can see the difference by changing your code slightly:

```python
my_list = list(range(1, 7))
my_list.append(8)
print(my_list)
```

When you run this code, it will successfully append the number eight to the end of your list.

Quiz 27 - f-string Number Bases

How well do you know your number bases? Wait! You didn't even realize there was more than one base!?

Most people are familiar with the base ten numeral system, also known as the decimal numeral system, but several others exist.

This chapter's quiz will challenge you to determine what a number is converted to in other numeral systems. Good luck!

The Quiz

What is the output of this code?

```python
number = 12
print(f"int: {number:d}; hex: {number:02X}; oct: {number:o}; bin: {number:b} ")
```

A. int: 12; hex: 0C; oct: 14; bin: 1100
B. int: c; hex: d; oct: 12; bin: 1100
C. int: 12; hex: c; oct: 12; bin: 1101
D. None of the above

Hint

If you get stuck, look up the other numeral systems. The string in the quiz has clues as to what they are called.

Answer 27 - f-string Number Bases

This is the answer to **Quiz 27 - f-string Number Bases**.

Your original question was:

What is the output of this code?

```python
number = 12
print(f"int: {number:d}; hex: {number:02X}; oct: {number:o}; bin: {number:b} ")
```

A. int: 12; hex: 0C; oct: 14; bin: 1100
B. int: c; hex: d; oct: 12; bin: 1100
C. int: 12; hex: c; oct: 12; bin: 1101
D. None of the above

The Answer

A) int: 12; hex: 0C; oct: 14; bin: 1100

Explanation

If you have been doing the quizzes in order, you will know that whenever you see a colon inside the curly braces in an f-string, you will be doing some special formatting to that variable.

In this case, you are formatting the input number into one numeral system or another. This quiz does the following numeral systems:

- decimal (10-digit)
- hexadecimal (16-digit)
- octal (8-digit)
- binary (2-digit)

It is outside the scope of this book to teach you four separate numeral systems.

You can run the number **12** through various conversion tools on the web to verify that it converts to the numerals in the answer above, though.

Check out the following websites for more information:

- Zetcode - Python f-string[28]
- Real Python - Python 3's f-Strings: An Improved String Formatting Syntax (Guide)[29]

[28]https://zetcode.com/python/fstring/
[29]https://realpython.com/python-f-strings/

Quiz 28 - Boolean Shenanigans

You have seen a couple of quizzes already that used Python Booleans. However, you haven't gotten much practice with the built-in `bool()` function.

Now's your chance to show what you know about `bool()`. You will use the `bool()` function in this quiz three times.

Can you figure out the answer?

The Quiz

What is the output of this code?

```python
print(bool(bool((lambda a, b: a * b)(4, 5)) - bool(True)))
```

A. 19
B. 0
C. True
D. False

Hint

Read this code very carefully.

Answer 28 - Boolean Shenanigans

This is the answer to **Quiz 28 - Boolean Shenanigans**.

Your original question was:

What is the output of this code?

```
print(bool(bool((lambda a, b: a * b)(4, 5)) - bool(True)))
```

A. 19
B. 0
C. True
D. False

The Answer

D) False

Explanation

Removing the nesting is the best way to figure out nested function calls. You will find that is the quickest way to figure out this quiz.

The first step is to pull the `lambda` out:

```
>>> (lambda a, b: a * b)(4, 5)
20
```

The result of the `lambda` call is 20. Now you can rewrite that inner `bool()` call:

```
>>> bool(20)
True
```

Calling `bool()` on non-zero integers will always return `True`. Now you can rewrite the quiz question like this:

```python
print(bool(True - bool(True)))
```

The code above subtracts `True` from `True`. Python Booleans are a subclass of `int` where `True` maps to one.

Let's rewrite the code one more time:

```python
print(bool(1 - 1))
```

One minus one is zero. Passing zero to the `bool()` function will return `False`, which is the answer to this quiz!

Quiz 29 - String Slicing Silliness

Slicing is a term in Python that refers to creating a subset of an iterable. You can use slicing to create smaller lists and smaller strings.

This quiz challenges your knowledge of the slicing syntax. What happens when you supply only some of the arguments? What does a negative one do when you're slicing?

Good luck!

The Quiz

What prints out when you run this code?

```python
print("Python"[::-1])
```

A. An exception occurs
B. n
C. nohtyP
D. Pytho

Hint

Strings support the same slicing syntax as Python lists do.

Answer 29 - String Slicing Silliness

This is the answer to **Quiz 29 - String Slicing Silliness**.

Your original question was:

What prints out when you run this code?

```
print("Python"[::-1])
```

A. An exception occurs
B. n
C. nohtyP
D. Pytho

The Answer

C) nohtyP

Explanation

The strange syntax in this quiz is a shorthand way of telling Python to reverse a string.

However, that is a boring explanation, so let's look at how you might find the other answers here too!

To get the last letter of a string, you would use this syntax:

```
>>> "Python"[-1]
'n'
```

If you want to get all the letters EXCEPT the last one, you would need to do this:

```
>>> "Python"[0:-1]
'Pytho'
```

When you see [::-1], that tells Python to start at the end of the string and count down to the beginning while stepping backward one character at a time. Since you are not specifying the start or stop, Python goes through the whole string.

Take a look at the following examples to see how it changes by specifying different stop increments:

```
>>> "Python"[:4:-1]
'n'
>>> "Python"[:3:-1]
'no'
>>> "Python"[:2:-1]
'noh'
>>> "Python"[:1:-1]
'noht'
>>> "Python"[:0:-1]
'nohty'
>>> "Python"[::-1]
'nohtyP'
```

Quiz 30 - Enumerate and Dictionaries

Python's `enumerate()` function is an excellent way to loop over an iterable. You use `enumerate()` to keep track of where you are while iterating.

But what happens when you use`enumerate()` with Python's `dict()` function?

You will need to think about this topic logically and see if you can come up with a solution!

The Quiz

What is the output of this weird code?

```python
# 30_enumerate_and_dict.py

some_string = "Python"
some_dict = {}
for i, some_dict[i] in enumerate(some_string):
    i = 10

print(some_dict)
```

A. TypeError
B. `{}`
C. `{0: 'P', 1: 'y', 2: 't', 3: 'h', 4: 'o', 5: 'n'}`
D. None of the above

Hint

What does `enumerate()` do?

Answer 30 - Enumerate and Dictionaries

This is the answer to **Quiz 30 - Enumerate and Dictionaries**.

Your original question was:

What is the output of this weird code?

```python
# 30_enumerate_and_dict.py

some_string = "Python"
some_dict = {}
for i, some_dict[i] in enumerate(some_string):
    i = 10

print(some_dict)
```

A. TypeError
B. `{}`
C. `{0: 'P', 1: 'y', 2: 't', 3: 'h', 4: 'o', 5: 'n'}`
D. None of the above

The Answer

C) **`{0: 'P', 1: 'y', 2: 't', 3: 'h', 4: 'o', 5: 'n'}`**

Explanation

The `enumerate()` function returns a tuple containing a count and the values from the iterable.

Here's an example using the string from the quiz:

```
some_string = "Python"

for i, value in enumerate(some_string):
    print(f"{i = } {value = }")

i = 0 value = 'P'
i = 1 value = 'y'
i = 2 value = 't'
i = 3 value = 'h'
i = 4 value = 'o'
i = 5 value = 'n'
```

Now try rewriting the loop using the dictionary instead:

```
for i, some_dict[i] in enumerate(some_string):
    print(f"{some_dict= }")

some_dict= {0: 'P'}
some_dict= {0: 'P', 1: 'y'}
some_dict= {0: 'P', 1: 'y', 2: 't'}
some_dict= {0: 'P', 1: 'y', 2: 't', 3: 'h'}
some_dict= {0: 'P', 1: 'y', 2: 't', 3: 'h', 4: 'o'}
some_dict= {0: 'P', 1: 'y', 2: 't', 3: 'h', 4: 'o', 5: 'n'}
```

What's going on here? In the first iteration, "i" is zero and you implicitly pass the value to the dictionary when you call `some_dict[i]`. The first time through the loop, this is the same as saying:

```
some_dict[0] = "P"
```

Each time you iterate a letter in the string, you set another integer as the key and another string as its value.

Quiz 31 - Dynamic Docstrings

Python supports the ability to add a value to a variable and assign the new value back to it using the += operator. The += operator is sometimes called **addition assignment** or **augmented assignment**.

Python supports several different kinds of augmented assignment:

- +=
- -=
- *=
- /=
- %=

If you understand how this works, you have a leg up on the competition for this fun quiz!

The Quiz

What does the following code output?

```python
header = "Say hello in Python"

def hello(name: str) -> str:
    """
    %s

    Python is amazing!
    """
    return f"Hello {name}. Nice to meet you!"

hello.__doc__ %= header
help(hello)
```

A. An exception is raised

B)
Help on function hello in module `__main__`:

hello(name: str) -> str

Say hello in Python

Python is amazing!

C)
Help on function hello in module `__main__`:

hello(name: str) -> str

%s

Python is amazing!

D. None of the above

Hint

Augmented assignment works with some non-numeric data types.

Answer 31 - Dynamic Docstrings

This is the answer to **Quiz 31 - Dynamic Docstrings**.

Your original question was:

What does the following code output?

```python
header = "Say hello in Python"

def hello(name: str) -> str:
    """
    %s

    Python is amazing!
    """
    return f"Hello {name}. Nice to meet you!"

hello.__doc__ %= header
help(hello)
```

A. An exception is raised

B)
Help on function hello in module `__main__`:

hello(name: str) -> str

Say hello in Python

Python is amazing!

C)
Help on function hello in module `__main__`:

hello(name: str) -> str

%s

Python is amazing!

D. None of the above

The Answer

B)
Help on function hello in module `__main__`:

hello(name: str) -> str

Say hello in Python

Python is amazing!

Explanation

Python supports three different methods of string interpolation. Here they are:

- Using the %s method (the oldest)
- Using the `format()` string method
- Using f-strings

Here are examples of using each of these methods:

```
>>> name = "Mike"
>>> "Hello %s" % name
'Hello Mike'
>>> "Hello {}".format(name)
'Hello Mike'
>>> f"Hello {name}"
'Hello Mike'
```

Python functions have attributes. If you run `dir()` on the `hello()` function, you will see the following:

```
>>> dir(hello)
['__annotations__',
 '__call__',
 '__class__',
 '__closure__',
 '__code__',
 '__defaults__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__get__',
 '__getattribute__',
 '__globals__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__kwdefaults__',
 '__le__',
 '__lt__',
 '__module__',
 '__name__',
 '__ne__',
 '__new__',
 '__qualname__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__']
```

In the quiz, you access the `hello()` function's `__doc__` attribute, which defines the function's docstring. It would be best if you examined the docstring closely so that you notice that it has a `%s` inside of it. That `%s` tells Python that you can insert a string at that spot in the docstring.

The following code: `hello.__doc__ %= header` is equivalent to `hello.__doc__ = hello.__doc__ % header`.

Regardless of how you write the code, you are telling Python that you want to insert the `header` string into the `hello()` function's docstring.

Quiz 32 - Magical Multiplication

Python supports multiplication using the asterisk operator. Multiplication is a standard feature in all programming languages.

Here is how you would do multiplication in Python:

```
>>> 5 * 6
30
>>> a = 2
>>> b = 5
>>> a * b
10
```

But do you know how multiplication works under the covers?

Let's see if you know your stuff!

The Quiz

What is the result when you run this code?

```
multipliers = {"macbook": 600 .__mul__,
               "microbit": 5 .__mul__,
               "pico": 50 .__mul__}
print(multipliers["microbit"](6))
```

A. Syntax Error
B. `5 .__mul__`
C. 30
D. None of the above

Hint

Everything in Python is an object!

Answer 32 - Magical Multiplication

This is the answer to **Quiz 32 - Magical Multiplication**.

Your original question was:

What is the result when you run this code?

```
multipliers = {"macbook": 600 .__mul__,
               "microbit": 5 .__mul__,
               "pico": 50 .__mul__}
print(multipliers["microbit"](6))
```

A. Syntax Error
B. `5 .__mul__`
C. 30
D. What is this madness?

The Answer

C) 30

Explanation

Everything in Python is an object, including integers. However, Python's parser thinks that an integer with a period is a floating-point number and will parse it as such. You must put a space between the integer and the dunder method to get around that.

If that object has one, anything in Python that uses an asterisk for multiplication will call the `__mul__` dunder method. In this quiz, you don't need to use an asterisk. Instead, you can call the `__mul__` method directly by accessing a dictionary key and passing it a value. This syntax is sometimes called **dictionary dispatch**.

What that means is that this code:

```
>>> multipliers["microbit"](6)
30
```

is the equivalent of this code:

```
>>> 5 .__mul__(6)
30
```

It would be best if you tried running `dir()` on an integer to see what else you can do with them:

```
dir(5)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
 '__delattr__',  '__dir__', '__divmod__', '__doc__', '__eq__',
 '__float__', '__floor__', '__floordiv__', '__format__', '__ge__',
 '__getattribute__', '__getnewargs__', '__gt__', '__hash__', '__index__',
 '__init__', '__init_subclass__', '__int__', '__invert__', '__le__',
 '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__',
 '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__',
 '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__',
 '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__',
 '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
 '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__',
 '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__',
 'as_integer_ratio', 'bit_length', 'conjugate', 'denominator',
 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

That’s a lot of methods and attributes!

Quiz 33 - Deranged Generators

Python has the concept of iterators and generators. A generator is a function that behaves like an iterator. In other words, a generator function allows you to loop over a function!

You can use a generator to lazily loop over large files, make many database queries, and more.

Knowing how to use a generator can be a valuable way to avoid buffer overflows and application crashes.

You will need to use your knowledge of generators to solve this quiz. Can you do it? Let's find out!

The Quiz

What is the output of this crazy code?

```python
array = [21, 49, 15]
gen = (x for x in array if array.count(x) > 0)
array = [0, 49, 88]
print(list(gen))
```

A. [21, 49, 15]
B. [0, 49, 88]
C. [49]
D. None of the above

Hint

You should brush up on generator expressions.

Answer 33 - Deranged Generators

This is the answer to **Quiz 33 - Deranged Generators**.

Your original question was:

What is the output of this crazy code?

```python
array = [21, 49, 15]
gen = (x for x in array if array.count(x) > 0)
array = [0, 49, 88]
print(list(gen))
```

A. [21, 49, 15]
B. [0, 49, 88]
C. [49]
D. None of the above

The Answer

C) [49]

Explanation

The code in this quiz is tricky! Here you are modifying the list that the generator wants to use.

Here is the key to understanding what is happening:

- The `for` loop uses the first `array`
- The `if` statement uses the second array

The reason for this oddity is that the conditional statement is late binding.

If you modify the code a bit, you can see what is happening:

```python
array = [21, 49, 15]
gen = ((x, print(x, array)) for x in array)
array = [0, 49, 88]
```

When you run this code, you will get the following output:

```
21 [0, 49, 88]
49 [0, 49, 88]
15 [0, 49, 88]
```

The output above shows you that the `for` loop is iterating over the original `array`, but the conditional statement checks the newer array.

The only number that matches from the original array to the new array is **49**, so the count is one, which is greater than zero. That's why the output only contains `[49]`!

Quiz 34 - What Makes a Function?

Here is a short and easy quiz compared to the last one. But it's good to have various difficulty levels as it gives you many chances to get some (or all) of the quizzes right.

Do you know what makes up a function in Python? You'll find this quiz straightforward if you know all the required syntax.

If not, then you may be thumping your head against the wall!

The Quiz

What's wrong with this minimal function?

```python
def function():
    # This is a function

function()
```

A. Functions don't need parentheses!
B. You need a docstring, not a comment
C. You need a statement/expression in the function body
D. B or C
E. None of the above

Hint

Read up on how functions work. Check out Python 101 - An Intro to Functions[30] if you need some help!

[30]https://www.blog.pythonlibrary.org/2022/06/28/python-101-an-intro-to-functions/

Answer 34 - What Makes a Function?

This is the answer to **Quiz 34 - What Makes a Function?**.

Your original question was:

What's wrong with this minimal function?

```python
def function():
    # This is a function

function()
```

A. Functions don't need parentheses!
B. You need a docstring, not a comment
C. You need a statement/expression in the function body
D. B or C
E. None of the above

The Answer

D) B or C

Explanation

Wait a minute! There are two correct answers? Why yes, there are.

A valid minimal function can take the following forms:

- A function with only a docstring

```python
def function():
    """This is a function"""
```

- A function with only a statement (any statement will do) in it:

```python
def function():
    print
```

- A function with just `pass` in it:

```python
def function():
    pass
```

- A function with an ellipsis in it:

```python
def function():
    ...
```

These are all valid, minimal functions. They don't do anything or return anything, but they are valid Python code!

Quiz 35 - More Fun with Walruses

Walruses are not only amazing sea mammals; they are also the nickname of a particular syntax type in Python. The walrus operator (i.e. `:=`) allows you to assign a variable inside an expression.

You can use a walrus operator where repeated function calls can make your code slower or where repeated statements can make your code unwieldy.

But enough chatter about walrus operator use cases. Now is the time to take the quiz and test your knowledge!

The Quiz

What does the following code print out when you run it?

```python
(x := [1, 2, 3]).extend(x)
print(x)
```

A. SyntaxError
B. TypeError
C. [1, 2, 3]
D. [1, 2, 3, 1, 2, 3]

Hint

PEP 572 – Assignment Expressions[31] can help you.

[31]https://peps.python.org/pep-0572/

Answer 35 - More Fun with Walruses

This is the answer to **Quiz 35 - More Fun with Walruses**.

Your original question was:

What does the following code print out when you run it?

```
(x := [1, 2, 3]).extend(x)
print(x)
```

A. SyntaxError
B. TypeError
C. [1, 2, 3]
D. [1, 2, 3, 1, 2, 3]

The Answer

D) [1, 2, 3, 1, 2, 3]

Explanation

The walrus operator allows you to assign a value to a variable in an expression. The expression in this quiz is what is inside the parentheses.

PEP 572 – Assignment Expressions[32] calls this syntax "valid, though not recommended" because it is unintuitive.

A somewhat better way to write the code would be the following:

```
>>> (x := [1, 2, 3])
[1, 2, 3]
>>> x.extend(x)
>>> x
[1, 2, 3, 1, 2, 3]
```

Here you assign the list `[1, 2, 3]` to the variable `x`. You still use the discouraged syntax, but it is slightly more readable.

Your next step is to extend the list using that same list.

The following code is equivalent:

[32] https://peps.python.org/pep-0572/

```
>>> (x := [1, 2, 3])
[1, 2, 3]
>>> x.extend([1, 2, 3])
>>> x
[1, 2, 3, 1, 2, 3]
```

Quiz 36 - Dictionary Comprehension Trickery

Python supports several different kinds of comprehensions. The most popular comprehension is the list comprehension, a one-liner that allows you to create a list while iterating.

Another popular type of comprehension is the dict comprehension! As the name implies, you use the dict comprehension to create Python dictionaries!

This quiz will test your knowledge of how the dict comprehension is created. Is this the correct syntax? If it is, what does the dictionary look like?

Let's find out!

The Quiz

What is the output of the following code snippet?

```python
print({key: value for key, value in enumerate("abcde")})
```

A. `{}`
B. Syntax Error
C. `{0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e'}`
D. None of the above

Hint

PEP 274[33] is your friend.

[33] https://peps.python.org/pep-0274/

Answer 36 - Dictionary Comprehension Trickery

This is the answer to **Quiz 36 - Dictionary Comprehension Trickery**.

Your original question was:

What is the output of the following code snippet?

```python
print({key: value for key, value in enumerate("abcde")})
```

A. `{}`
B. Syntax Error
C. `{0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e'}`
D. None of the above

The Answer

C) **`{0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e'}`**

Explanation

The first step in understanding this quiz is to grasp what the `enumerate()` function is returning.

You can rewrite part of the code to figure that out by doing the following:

```python
>>> for key, value in enumerate("abcde"):
...     print(f"{key = } {value = }")
...
key = 0 value = 'a'
key = 1 value = 'b'
key = 2 value = 'c'
key = 3 value = 'd'
key = 4 value = 'e'
```

Here you can see that `enumerate()` returns the index of each item it iterates over in addition to the value. In this example, you name the index variable `key` and the other variable `value`.

You write a dict comprehension to create **key**: **value** pairs for each iteration over the iterator. By using `enumerate()`, you get your key and value pairs automatically! The keys are the indexes of each item in the iterable and the values are each item in the iterable.

Quiz 37 - Multiple Assignment Mania

Do you know how multiple assignment works in Python? Is it allowed? What is the syntax for multiple assignment?

These are great questions! You will need to know the answer to at least one of these questions to solve this riddle!

Here it is:

The Quiz

What is the output of this silly code?

```python
d = 0
a = b = c = d
print(a)
```

A. NameError
B. 0
C. TypeError
D. None of the above

Hint

If you get stuck, look up the term "multiple assignment" to learn more.

Answer 37 - Multiple Assignment Mania

This is the answer to **Quiz 37 - Multiple Assignment Mania**.

Your original question was:

What is the output of this silly code?

```
d = 0
a = b = c = d
print(a)
```

A. NameError
B. 0
C. TypeError
D. None of the above

The Answer

B) 0

Explanation

What is this coding paradigm called anyway? `a = b = c = d` looks weird!

These consecutive equal signs mean that you want to set `a`, `b`, and `c` as equal to `d`.

The code in this quiz assigns the same value to multiple variables. But there is another type of multiple assignment:

```
>>> name, job = "Mike", "Python Wrangler"
>>> name
'Mike'
>>> job
'Python Wrangler'
```

In this example, you assigned the two strings on the right of the assignment operator (i.e., the equals sign) to the two variables on the left (i.e., `name` and `job`).

You will probably see this latter type of multiple assignment more often than the former in production code or examples on the Internet.

Quiz 38 - Exhausting Generators

Python generators give you an excellent way to iterate over large data sets a chunk at a time. The generator prevents you from running out of memory.

There are several ways to create generators in Python. The most well-known method is to create a function and yield a value rather than return it.

Do you know what happens when you ask a generator for more values than it contains? Prove it!

The Quiz

What is the output when you call `next()` three times on this generator?

```
>>> def numbers():
...     yield 1
...     yield 2
...
>>> x = numbers()
>>> next(x)
1
>>> next(x)
2
>>> next(x)
???
```

A. 1
B. 2
C. RuntimeError
D. StopIteration

Hint

If you're lost, you'll want to look up the term "exhausting a generator."

Answer 38 - Exhausting Generators

This is the answer to **Quiz 38 - Exhausting Generators**.

Your original question was:

What is the output when you call `next()` three times on this generator?

```
>>> def numbers():
...     yield 1
...     yield 2
...
>>> x = numbers()
>>> next(x)
1
>>> next(x)
2
>>> next(x)
???
```

A. 1
B. 2
C. RuntimeError
D. StopIteration

The Answer

D) StopIteration

Explanation

The `numbers()` generator function has two `yield` statements. The `yield` statements indicate that the generator will yield two integers. When a generator reaches the end of its values, it is said to be "exhausted".

You can iterate over `numbers()` (or exhaust it) in several ways.

Here's how you would do it with a `for` loop:

```
>>> def numbers():
...     yield 1
...     yield 2
...
>>> for number in numbers():
...     print(number)
...
1
2
```

You can also use Python's `list()` function to transform a generator directly into a list:

```
>>> def numbers():
...     yield 1
...     yield 2
...
>>> list(numbers())
[1, 2]
```

Note that both the `for` loop and the `list()` function do not raise the `StopIteration` exception. The reason is that both `list()` and the `for` loop know what to do when they reach the end of the generator.

In the quiz above, you did not wrap the calls to `next()` in an exception handler. When you attempt to extract the third value from the two-value generator, Python raises the `StopIteration` exception to let you know there are no more items.

A related topic is the `__iter__()` magic method. To learn more about creating iterators, check out the following resource:

- StackOverflow - How to build a basic iterator?[34]

[34]https://stackoverflow.com/a/7542261/208880

Quiz 39 - A List O' Functions

Functions are "first-class objects" in Python. That means that functions behave like any other object, such as an integer or a string. You can nest functions inside each other, pass a function around to another function, and more.

Now that you know that, you can figure out what's going on in this quiz.

Let's find out!

The Quiz

What will the output of the following code be?

```python
functions = []

for number in range(8):
    def my_function():
        return number
    functions.append(my_function)

results = [function() for function in functions]
print(results)

"""
A) [<function my_function at 0x7f842c7218b0>,
    <function my_function at 0x7f842c7218b0>,
    <function my_function at 0x7f842c7218b0>,
    <function my_function at 0x7f842c7218b0>
    <function my_function at 0x7f842c7218b0>,
    <function my_function at 0x7f842c7218b0>,
    <function my_function at 0x7f842c7218b0>,
    <function my_function at 0x7f842c7218b0>]
    <function my_function at 0x7f842c7218b0>,
    <function my_function at 0x7f842c7218b0>]
B) [1, 2, 3, 4, 5, 6, 7, 8]
C) [0, 1, 2, 3, 4, 5, 6, 7]
D) [7, 7, 7, 7, 7, 7, 7, 7]
E) [8, 8, 8, 8, 8, 8, 8, 8]
"""
```

Hint

When is the function called?

Answer 39 - A List O' Functions

This is the answer to **Quiz 39 - A List O' Functions**.

Your original question was:

What will the output of the following code be?

```python
functions = []

for number in range(8):
    def my_function():
        return number
    functions.append(my_function)

results = [function() for function in functions]
print(results)

"""
A) [<function my_function at 0x7f842c7218b0>,
    <function my_function at 0x7f842c7218b0>,
    <function my_function at 0x7f842c7218b0>,
    <function my_function at 0x7f842c7218b0>
    <function my_function at 0x7f842c7218b0>,
    <function my_function at 0x7f842c7218b0>,
    <function my_function at 0x7f842c7218b0>,
    <function my_function at 0x7f842c7218b0>]
    <function my_function at 0x7f842c7218b0>,
    <function my_function at 0x7f842c7218b0>]
B) [1, 2, 3, 4, 5, 6, 7, 8]
C) [0, 1, 2, 3, 4, 5, 6, 7]
D) [7, 7, 7, 7, 7, 7, 7, 7]
E) [8, 8, 8, 8, 8, 8, 8, 8]
"""
```

The Answer

D) [7, 7, 7, 7, 7, 7, 7, 7]

Explanation

A quick way to determine how this code works is to deconstruct it.

Try re-writing the code to look like this:

```
>>> for number in range(8):
...     print(number)
...
0
1
2
3
4
5
6
7
>>> number
7
```

When you iterate over a range, by default, the range will start at zero and end at the number provided minus one. In this case, you give the number eight, so the result is that you iterate over eight numbers, zero through seven (0-7).

You append `my_function` to the `functions` list in each iteration. Note that you do NOT call the function when you are appending it.

Instead, you call ALL the function instances in the **list comprehension** that follows the `for` loop.

What is the value of `number` after you have exited the `for` loop? It's seven (7)! You call eight `my_function` instances, and they ALL return seven!

Quiz 40 - Fixing the Math

One of the best ways to learn to code is to fix broken code. You will often find yourself working on bugs created by yourself or others in your organization.

Software bugs are only sometimes obvious. Sometimes the code will work fine in most cases, but you will have edge cases that take your application down.

This quiz will help you learn how to fix your code!

The Quiz

How do you get the following code to work? It should take in a number, return the square root and print it.

```
from math import sq_root


def get_the_root(number);
    return sq_root(number)


if name == main:
    get_the_root(64)
```

Hint

You will need to know a little about the `math` module, how functions are defined, and something else too!

Answer 40 - Fixing the Math

This is the answer to **Quiz 40 - Fixing the Math**.

Your original question was:

How do you get the following code to work? It should take in a number, return the square root and print it.

```python
from math import sq_root


def get_the_root(number);
    return sq_root(number)


if name == main:
    get_the_root(64)
```

The Answer

Here is one answer that will make the code work properly:

```python
# 40_fixing_the_math.py

from math import sqrt


def get_the_root(number):
    return sqrt(number)


if __name__ == "__main__":
    print(get_the_root(64))
```

Explanation

What was wrong with the original code anyway?

There are problems on ALL of the lines:

```python
from math import sq_root  # 1 <-- Bad import


def get_the_root(number);   # 2 <-- Function not created correctly
    return sq_root(number)  # 3 <-- Incorrect call


if name == main:  # 4 <-- Incorrect conditional
    get_the_root(64)  # 5 <-- Doesn't print out the returned value
```

First of all, the import is wrong! There is no `sq_root()` function in Python's `math` module. What you want is the `sqrt()` function instead.

Second, when you create the function, the `def` line must end in a colon, NOT a semi-colon!

Third, you call the wrong `math` function, so when you fix the import at the top, you'll need to update this line too!

Fourth, you must write the conditional statement like this:

```python
if __name__ == "__main__":
```

StackOverflow[35] has a couple of great explanations for how this code works and why you would use it.

Fifth, you aren't printing the result of the call to `get_the_root()`.

Other solutions will fix all the code in the quiz.

The following example removes the `get_the_root()` function entirely, for example:

```python
from math import sqrt as get_the_root


if __name__ == "__main__":
    print(get_the_root(64))
```

Here you alias the `math` module's `sqrt()` function to `get_the_root()`. Now that you have that alias, you can drop the custom function entirely!

[35] https://stackoverflow.com/questions/419163/what-does-if-name-main-do

Quiz 41 - Runaway Characters

Junior engineers and developers learn a lot through their mistakes. You also learn a lot by simply doing.

What does that mean, though? It means that sometimes you have to try out multiple solutions before you find one that works the way you want it to. Then you sometimes need to iterate on that working solution to optimize it or harden it against bad data.

Your challenge today is to analyze a string and figure out if it's a normal string!

The Quiz

What is the output of this code?

```python
path = "C:\Program Files\turtle game"
print(path)
```

A. RuntimeError
B. C:Program Files\turtle game
C. C:Program Files urtle game
D. None of the above

Hint

You may want to brush up on your escape characters.

Answer 41 - Runaway Characters

This is the answer to **Quiz 41 - Runaway Characters**.

Your original question was:

What is the output of this code?

```python
path = "C:\Program Files\turtle game"
print(path)
```

A. RuntimeError
B. C:Program Files\turtle game
C. C:Program Files urtle game
D. None of the above

The Answer

C) C:Program Files urtle game

Explanation

Programming languages need to have a way to insert "illegal" characters into a string. You can't see an illegal character printed, such as a tab or a new line.

These characters are called **escape characters**. To create an escape character in Python, you must start with a backslash (\) followed by the character you want to insert.

Here are the most common escape characters in Python:

- `\'` Single Quote
- `\\` Backslash
- `\n` New Line
- `\r` Carriage Return
- `\t` Tab
- `\b` Backspace
- `\f` Form Feed

Microsoft Windows uses backslashes to separate parts of a file or folder path. These backslashes can make valid-looking paths invalid to Python.

One of the simplest solutions to this issue is to use a raw-string for a path in Windows:

```
path = r"C:\Program Files\turtle game"
```

The `r` pre-pended to the string tells Python that this is a raw-string, meaning that Python shouldn't try to escape any characters.

You can also escape an escape character by using an extra backslash:

```
>>> path = "C:\\Program Files\\turtle game"
>>> print(path)
C:\Program Files\turtle game
```

The general recommendation is to use double-backslashes in Windows paths for all the parts of the path.

Alternatively, you can use Python's great `pathlib` or `os.path` to join the parts of the paths together for you programmatically and let Python figure out which file system delimiter it should use.

Quiz 42 - Truthy or Falsey Take Two

What is `True` and what is `False` in Python? The Boolean values `True` and `False` are pretty explicit, but did you know that one is True and zero is False too?

Here are some other `True` and `False` conditions:

- `True` - A list with one or more items
- `False` - An empty string
- `True` - A dictionary with one or more keys and values
- `False` - An empty tuple

You can also test for a series of `True` conditions using the `all()` function.

Let's find out if you know how that works!

The Quiz

What is the output of the last line of code in this REPL session?

```
>>> all([True, True, True])
True
>>> all([True, True, False])
False

>>> print(all([[]]))
???
```

A. True
B. False
C. None of the above

Hint

See quiz #6!

Answer 42 - Truthy or Falsey Take Two

This answers **Quiz 42 - Truthy or Falsey Take Two**.

Your original question was:

What is the output of the last line of code in this REPL session?

```
>>> all([True, True, True])
True
>>> all([True, True, False])
False

>>> print(all([[]]))
???
```

A. True
B. False
C. None of the above

The Answer

B) False

Explanation

Do you remember what is `True` or `False` in Python?

Let's review:

- `True` - A list with one or more items
- `False` - An empty string
- `True` - A dictionary with one or more keys and values
- `False` - An empty tuple

In Python, an empty list is "Falsey". Try adding some extra spaces to the code example in the quiz:

```python
print(all( [ [] ] ))
```

You are asking Python's `all()` function to evaluate a single falsey item. This is equivalent to `all([False])`, which also returns `False`.

If you want it to be `True`, then add ANOTHER empty list to it: `all([ [ [] ] ])`. When you add that inner empty list, you are telling Python to evaluate a list with a single item inside of it, which happens to be an empty list. But because the outer list has an item, that list is NOT empty, so `all()` returns `True`.

Quiz 43 - More f-string Formatting Fun

You've done several quizzes on Python's f-strings already. But it's always good to review.

In this quiz, you will review how to format and align a string.

Let's see what you've got!

The Quiz

What is the output of this crazy code?

```python
align = "^"
fill_char = "*"
width = 25
snake = "Python"
print(f"{snake:{fill_char}{align}{width}}")
```

A. SyntaxError
B. `*******************Python`
C. `Python*******************`
D. `*********Python**********`

Hint

Review some of the previous quizzes you have done about f-strings.

Answer 43 - More f-string Formatting Fun

This answers **Quiz 43 - More f-string Formatting Fun**.

Your original question was:

What is the output of this crazy code?

```python
align = "^"
fill_char = "*"
width = 25
snake = "Python"
print(f"{snake:{fill_char}{align}{width}}")
```

A. SyntaxError
B. `*******************Python`
C. `Python*******************`
D. `*********Python**********`

The Answer

D. `*********Python**********`

Explanation

The answer to this quiz is technically in **Quiz 16** where the `^` character is explained. To save you from flipping back to that quiz though, the `^` character tells Python that the sub-string should align in the center of the 25-character string.

You use the `fill_char` to fill in all the other characters in the 25-character string that is not taken up by the sub-string.

Let's re-cap how to do alignment in an f-string:

- `<` - You want the sub-string left-aligned
- `>` - You want the sub-string right-aligned
- `^` - You want the sub-string centered

Play around with the alignment character to see how it changes the result.

Quiz 44 - Fun with Types

Python comes with lots of built-in functions. Some of these functions allow you to introspect your code, such as `dir()`, `help()` and `type()`. Introspection is a great tool to learn about unfamiliar code or new Python packages.

Your challenge today is to prove your knowledge of how the `type()` function works.

Can you beat this quiz? Only time will tell!

The Quiz

Which type does this code find?

```
print(type({*{}}))
```

A. dict
B. set
C. tuple
D. list

Hint

Watch out for that asterisk!

Answer 44 - Fun with Types

This is the answer to **Quiz 44 - Fun with Types**.

Your original question was:

Which type does this code find?

```
print(type({*{}}))
```

A. dict
B. set
C. tuple
D. list

The Answer

B) set

Explanation

What's going on here anyway?

The best way to discover what's happening is to open up a REPL session and try out the code:

```
>>> print(type({{}}))  # no asterisk
Traceback (most recent call last):
  Python Shell, prompt 3, line 1
builtins.TypeError: unhashable type: 'dict'
```

Without the asterisk, Python thinks you are trying to create a dictionary. However, you can't create a dictionary with a single empty dictionary inside of it. If you try that, you will get a `TypeError`!

However, when you use an asterisk on an empty dictionary, it unpacks the dictionary even though there's nothing to unpack!

You can see this happen if you create a function that lets you see the keyword arguments:

```
>>> def my_function(*args, **kwargs):
...     print(f"{args = } {kwargs = }")
...
>>> my_function(*{})
args = () kwargs = {}
```

When you unpack a dictionary, you may recall that unpacking operation returns keyword arguments. So in this example, you unpacked empty keyword arguments.

The lesson here is that if you unpack an empty dictionary inside curly braces, you create a set. Sets can be created with curly braces, too, after all:

```
>>> my_set = {1, 2}
>>> type(my_set)
<class 'set'>
```

That was a tricky one.

Quiz 45 - Boolean Equality

You have seen Boolean values used and abused in previous quizzes in this book. For this quiz, you'll use Booleans and the logical `not` operator to test for equality.

The question here is: What is the result?

Hopefully, you'll be able to glance at this code and know the answer!

The Quiz

What is the output when you run this code in the REPL?

```
>>> x = True
>>> y = False
>>> print(x == not y)
???
```

A. True
B. False
C. Syntax Error
D. None of the above

Hint

Always double-check your code in a quiz!

Answer 45 - Boolean Equality

This is the answer to **Quiz 45 - Boolean Equality**.

Your original question was:

What is the output when you run this code in the REPL?

```
>>> x = True
>>> y = False
>>> print(x == not y)
???
```

A. True
B. False
C. Syntax Error
D. None of the above

The Answer

C) Syntax Error

Explanation

This is a ridiculous quiz. It has a mistake in it!

Of course, you have to know a little bit about Python to see the mistake. Looking at code and diagnosing the problem is a LARGE part of becoming an effective software developer or engineer.

If you run this code in Python 3.10 or above, Python will try to tell you what's wrong with the code:

```
>>> x = True
>>> y = False
>>> print(x == not y)
  File "<stdin>", line 1
    print(x == not y)
               ^^^
SyntaxError: invalid syntax
```

There's something wrong with the `not` operator in the code above.

You could fix the code like this:

```
>>> x = True
>>> y = False
>>> print(x != y)
True
```

Here you use the `!=` to ask Python is `x` does not equal `y`.

The other way to "fix" this code is to add some extra parentheses:

```
>>> x = True
>>> y = False
>>> print(x == (not y))
True
```

In this example, you put "not y" in parentheses and compare that to `x`.

Both solutions are a good way to make the code work.

Quiz 46 - Manic Multistring

Python supports several different ways of creating strings. You can use any of the following:

- Single quotes
- Double quotes
- Triple quotes
- `str()`

But what happens when you mix and match different quote types? Is that supported as well?

If you know the answer, then this quiz is for you!

The Quiz

Is this valid code? True or False?

```python
print("Python""")
```

True or False?

Hint

Is this concatenation, bad syntax, or what?

Answer 46 - Manic Multistring

This is the answer to **Quiz 46 - Manic Multistring**.

Your original question was:

Is this valid code? True or False?

```python
print("Python""")
```

True or False?

The Answer

True

Explanation

You might be wondering what's going on with this crazy code.

Python is doing implicit concatenation here. It concatenates the following two strings:

- "Python"
- ""

Yes. That second pair of double quotes is an empty string.

You can prove this in the REPL by putting something in between that second pair of quotes:

```
>>> print("python" "is" "silly")
pythonissilly
>>> print("python""is""silly")
pythonissilly
```

The example above shows that you can separate the strings with spaces, and Python happily joins them together. You can skip the spaces, and Python *still* joins the three strings into one!

Quiz 47 - Slicing and Dicing

Python supports the concept of slicing. A slice is a smaller piece of a Python list, string, or tuple.

What is the syntax for creating a slice? Do you know?

The challenge of this quiz is to know the following:

- How the REPL works
- How to create a slice of a list
- Python's built-in functions

If you know all three, this will be a "slice of cake".

The Quiz

What happens when you run this code in a REPL?

```
>>> slice(1,3)
>>> my_list = [10, 3, 12, 30, 77]
>>> print(my_list[_])
```

A. Syntax Error
B. [10, 3, 12, 30, 77]
C. [10, 3]
D. [3, 12]

Hint

The only way to get the correct answer is to run this code in the REPL (and know what Python does with single underscores)!

Answer 47 - Slicing and Dicing

This is the answer to **Quiz 47 - Slicing and Dicing**.

Your original question was:

What happens when you run this code in a REPL?

```
>>> slice(1,3)
>>> my_list = [10, 3, 12, 30, 77]
>>> print(my_list[_])
```

A. Syntax Error
B. [10, 3, 12, 30, 77]
C. [10, 3]
D. [3, 12]

The Answer

D) [3, 12]

Explanation

Python has a LOT of different built-in functions[36]. Feel free to review them if you need to.

One of those built-in functions is `slice()`. Here is what the docs say about `slice()`:

```
Return a slice object representing the set of indices specified by range(start, stop\
, step).
```

In the REPL session in the code for this quiz, you can see that you don't assign the result of the `slice()` call, though. The result is deleted!

But not all is as it appears! In the REPL, Python will save the result of the last evaluated expression to a special variable: _.

You can see this in the following example:

[36]https://docs.python.org/3/library/functions.html

```
>>> slice(1,3)
slice(1, 3, None)
>>> my_list = [10, 3, 12, 30, 77]
>>> print(_)
slice(1, 3, None)
```

Here you print out the value of _ and you can see that it is the `slice()` object you created earlier. You can use the `slice()` object the same way you use regular slice syntax:

```
>>> slice(1,3)
slice(1, 3, None)
>>> my_list = [10, 3, 12, 30, 77]
>>> print(_)
slice(1, 3, None)
>>> my_list[_]
[3, 12]
>>> my_list[1:3]
[3, 12]
>>>
```

You do need to be careful with the underscore variable, though. If you try to use the underscore now, you may be surprised what it is:

```
>>> my_list[_]
Traceback (most recent call last):
  Python Shell, prompt 17, line 1
builtins.TypeError: list indices must be integers or slices, not list
>>> print(_)
[3, 12]
```

When you ran `my_list[1:3]`, the result of that call reassigned or overwrote the `slice()` object that was initially in the _ variable.

Quiz 48 - Text Finding Adventures

Everything in Python is an object, and every object has attributes, methods, or both.

Strings are no exception to this rule. This quiz will test your knowledge of the string's `find()` method.

What does `find()` do? What does `find()` return?

The Quiz

What is the output of this code?

```python
text = "Python is amazing!"
if text.find("C++"):
    print("Found C++!")
else:
    print("C++ not found!")
```

A. Found C++!
B. C++ not found!
C. None of the above

Hint

What is `True` or `False` in Python?

Answer 48 - Text Finding Adventures

This is the answer to **Quiz 48 - Text Finding Adventures**.

Your original question was:

What is the output of this code?

```python
text = "Python is amazing!"
if text.find("C++"):
    print("Found C++!")
else:
    print("C++ not found!")
```

A. Found C++!
B. C++ not found!
C. None of the above

The Answer

A) Found C++!

Explanation

What does the string's `find()` method return anyway? The quickest way to find out is to read the documentation[37].

According to the Python documentation, if the sub-string is **not** found, `find()` will return `-1`.

With that knowledge in hand, try running the following code:

```
>>> bool(-1)
True
```

When you cast `-1` to a Boolean value, it returns `True`.

In the quiz code above, when you **can't** find the sub-string, Python's `find()` method returns `-1`, which evaluates to `True` and then prints out: *Found C++!*

[37]https://docs.python.org/3/library/stdtypes.html#str.find

Quiz 49 - Puzzling Shadows

Sometimes it's fun to do a quiz with an open-ended answer instead of multiple choice.

For this quiz, you will attempt to make the code work. There is at least one thing wrong with the code in this quiz. Possibly more!

Can you make this code run and explain what the problem was?

The Quiz

How do you make this code work?

```python
# 49_puzzling_shadows.py

import math

def math(number):
    return math.sqrt(number)

def main(number):
    print(f"The square root of {number} is {math(number)}")

main(8)
```

Hint

If you get stuck, look up the term "shadowing imports."

Answer 49 - Puzzling Shadows

This is the answer to **Quiz 49 - Puzzling Shadows**.

Your original question was:

How do you make this code work?

```python
import math

def math(number):
    return math.sqrt(number)

def main(number):
    print(f"The square root of {number} is {math(number)}")

main(8)
```

The Answer

The primary answer is to not name any of your functions the same name as one of your imports.

Explanation

The problem with the code above is that your `math()` function is the same name as the imported `math` module. Python imports `math`, then it creates the `math()` function, which overwrites the module in memory.

This behavior is called "shadowing". You can shadow modules as well as some built-in functions in Python.

You should not shadow Python modules or built-in functions or classes, as this leads to strange bugs that can be difficult to understand and debug.

To fix the code above, you should rename your `math()` function to something else, such as `my_math()`:

```python
import math

def my_math(number):
    return math.sqrt(number)

def main(number):
    print(f"The square root of {number} is {my_math(number)}")

main(8)
```

Because this is a silly example, you can also do the same thing that you did in **Quiz 40** and remove your custom function entirely:

```python
import math

def main(number):
    print(f"The square root of {number} is {math.sqrt(number)}")

main(8)
```

Most good Integrated Development Environments (IDE), such as PyCharm or VS Code, should highlight an error like this. If you are programming with a text editor, it may not detect these types of errors.

Good luck, and keep coding!

Quiz 50 - A Print Puzzle

So you think you know how to use Python's `print()` function? That's great!

The `print()` function is one of the first things you learn about when learning the Python programming language.

Today you'll get to prove your knowledge of this useful function!

The Quiz

What is the output of this code?

```python
print("Python"  "is"    "amazing")
```

A. Python is amazing
B. Pythonisamazing
C. Syntax Error
D. None of the above

Hint

What parameters does the `print()` function take?

Answer 50 - A Print Puzzle

This is the answer to **Quiz 50 - A Print Puzzle**.

Your original question was:

What is the output of this code?

```
print("Python"  "is"    "amazing")
```

A. Python is amazing
B. Pythonisamazing
C. Syntax Error
D. None of the above

The Answer

B) Pythonisamazing

Explanation

The magic of the amazing and mysterious `print()` function doth baffle!

In this case, the string joining itself has *nothing* to do with the `print()` function.

Here's the proof:

```
>>> "Python"  "is"    "amazing"
'Pythonisamazing'
>>> a = "Python"  "is"    "amazing"
>>> a
'Pythonisamazing'
```

Python will implicitly join them all back together if you split up a string into multiple strings on the same line.

Python also lets you do crazy things like breaking up a long string using parentheses:

```
>>> a_long_string = ("The zebra jumped over the fox"
...                  "and ran into a zeppelin")
...
>>> a_long_string
'The zebra jumped over the foxand ran into a zeppelin'
```

Note that when it joins this string back together, it doesn't add a space between the two strings. You'll have to do that yourself!

Hopefully, you will use the tips you learned in this quiz in your code sometime soon!

Quiz 51 - The Ellipsis Condition

The Python programming language includes a special constant called the **Ellipsis**. You can create an Ellipsis object using . . .

The Ellipsis is used to signify an empty function or a kind of “TODO” for your code. You may also use the Ellipsis in slicing syntax.

Today you’ll get to prove your obscure knowledge of Python by solving this quiz!

The Quiz

What is the output of this code?

```python
age = 17

if age >= 18 or ...:
    print("I'm going to drive on my own!")
else:
    print("I need a driver!")
```

A. Syntax Error
B. Type Error
C. I’m going to drive on my own!
D. I need a driver!

Hint

Check out the documentation for the Ellipsis[38]

[38]https://docs.python.org/3/library/constants.html#Ellipsis

Answer 51 - The Ellipsis Condition

This is the answer to **Quiz 51 - The Ellipsis Condition**.

Your original question was:

What is the output of this code?

```python
age = 17

if age >= 18 or ...:
    print("I'm going to drive on my own!")
else:
    print("I need a driver!")
```

A. Syntax Error
B. Type Error
C. I'm going to drive on my own!
D. I need a driver!

The Answer

C) I'm going to drive on my own!

Explanation

The simple explanation here is that the Ellipsis is "truthy". Yes, it really is!

Take a look:

```python
>>> bool(...)
True
```

When you have something that evaluates to `True`, and you combine it with Python's logical `or` statement, then that condition will be `True`:

```
>>> bool(age >= 18 or ...)
True
>>> bool(age >= 18)
False
>>> bool(...)
True
```

Now you know an esoteric fact about Python too!

Quiz 52 - Exceptionally Crazy Again

Your very first quiz was about Python's exception handlers. That quiz used an example that is nearly identical to this quiz.

Your task is to determine what's different about this version and how it changes the output. Or does the edit change the output?

Good luck!

The Quiz

What is the output of this crazy exception handler?

```python
try:
    for i in range(3):
        try:
            1 / 0
        except ZeroDivisionError:
            raise ZeroDivisionError("Error: You divided by zero!")
        finally:
            print("Finally executed")
            return
except ZeroDivisionError:
    print("Outer ZeroDivisionError exception caught")
```

A. Outer ZeroDivisionError exception caught
B. Error: You divided by zero!
C. Error: You divided by zero!
   Finally executed
D. Finally executed
E. None of the above

Hint

Study the code closely to see what has changed!

Answer 52 - Exceptionally Crazy Again

This is the answer to **Quiz 52 - Exceptionally Crazy Again**.

Your original question was:

What is the output of this crazy exception handler?

```python
try:
    for i in range(3):
        try:
            1 / 0
        except ZeroDivisionError:
            raise ZeroDivisionError("Error: You divided by zero!")
        finally:
            print("Finally executed")
            return
except ZeroDivisionError:
    print("Outer ZeroDivisionError exception caught")
```

A. Outer ZeroDivisionError exception caught
B. Error: You divided by zero!
C. Error: You divided by zero!
   Finally executed
D. Finally executed
E. None of the above

The Answer

E) None of the above

Explanation

Wait! What? Why is the answer "E" here?

In Python, you cannot use the keyword, `return`, outside of a function. If you do, you'll get a Syntax Error like this: `Syntax Error: 'return' outside function:`

Try running the code above and see for yourself!

Quiz 53 - Crazy Mixed Up Code

It is time for another quiz exercise! Instead of giving you a multiple choice answer, you'll have to code to figure this one out!

Look over the quiz code carefully to find all the errors. Once you have found them all, write out your answer.

The answer in this book may differ from yours, but they should be similar.

Good luck!

The Quiz

Can you make this code work again?

```python
def type_checker(variable);
    if isinstance(variable, [float, integer]):
        print(f"{variable} is a number")
    elif isinstance(variable, string):
        print(f"{variable} is a string")
    elif ...:
        print("{variable} is something else".format(variable))
    else:
        break

if __name__ == "_main_":
    typ_checker(variable="Python")
```

Hint

Study the code and run it over and over with each edit. You'll get there!

Answer 53 - Crazy Mixed Up Code

This is the answer to **Quiz 53 - Crazy Mixed Up Code**.

Your original question was:

Can you make this code work again?

```python
def type_checker(variable);
    if isinstance(variable, [float, integer]):
        print(f"{variable} is a number")
    elif isinstance(variable, string):
        print(f"{variable} is a string")
    elif ...:
        print("{variable} is something else".format(variable))
    else:
        break

if __name__ == "_main_":
    typ_checker(variable="Python")
```

The Answer

Here is a potential answer:

```python
def type_checker(variable):  # <-- Replace semicolon with colon
    if isinstance(variable, (float, int)):  # <-- integer to int / list to tuple
        print(f"{variable} is a number")
    elif isinstance(variable, str):  # <-- Change string to str
        print(f"{variable} is a string")
    elif ...:
        print("{variable} is something else".format(variable))
    else:
        print("Default")   # Remove the break statement

if __name__ == "__main__":  # <-- Fix "_main_" to "__main__"
    type_checker(variable="Python")  # <-- Call the right function name
```

Explanation

There are a lot of silly errors in this quiz's code. Seven of them to be exact.

You will go over each of the errors so that you understand what was wrong:

Starting at the top, the first error is that there was a a semi-colon at the end of the `type_checker()` function definition:

```python
def type_checker(variable);   # <-- Semi-colon should be a colon here
```

The next issue is two-fold. When you want to use the `isinstance()` function to test against multiple types, the types must be listed in a Python `tuple`, not a `list` (parentheses vs square brackets). The other issue is that `integer` isn't a valid Python type. You should use `int` instead:

```python
# Incorrect
if isinstance(variable, [float, integer]):

# Correct
if isinstance(variable, (float, int)):
```

The fourth issue is similar to the last issue in that `string` is also not a valid Python data type. To check for a string, use `str`:

```python
# Incorrect
elif isinstance(variable, string):

# Correct
elif isinstance(variable, str):
```

The fifth issue is that you can't use a `break` statement outside of a `for` or `while`-loop.

```python
# Incorrect
else:
    break

# Correct
else:
    print("Default")
```

The `else` statement could even be removed altogether since it will never be reached (because of the `elif ...` above it).

The sixth issue is a simple typo where you must use `__main__` instead of `_main_` to make the conditional at the end of your code work:

```python
# Incorrect
if __name__ == "_main_":

# Correct
if __name__ == "__main__":
```

The last issue is that there is a typo when you attempt to call `type_checker()` at the end of your code:

```python
# Incorrect
typ_checker(variable="Python")

# Correct
type_checker(variable="Python")
```

If you fix all seven of the issues, the code will then work!

Quiz 54 - Return vs. Yield

Did you know that Python functions always return something even if you don't specify what that something is?

A Python function doesn't require using the `return` statement in functions or classes. Instead, you can use the `yield` statement to create a generator.

Starting in Python 3.3, you can have a `return` and a `yield` statement inside the same function!

But what happens when you have both? Find out in today's quiz!

The Quiz

What will be the output of the following code?

```python
def my_func(value):
    if value == 5:
        return ["Python"]
    else:
        yield from range(value)

print(list(my_func(5)))
```

A. ["Python"]
B. [0, 1, 2, 3, 4]
C. []
D. None of the above

Hint

If you're stuck, you may find a clue in PEP 380[39].

[39]https://peps.python.org/pep-0380/

Answer 54 - Return vs. Yield

This is the answer to **Quiz 54 - Return vs Yield**.

Your original question was:

What will be the output of the following code?

```python
def my_func(value):
    if value == 5:
        return ["Python"]
    else:
        yield from range(value)

print(list(my_func(5)))
```

A. [“Python”]
B. [0, 1, 2, 3, 4]
C. []
D. None of the above

The Answer

C) []

Explanation

The first step in understanding what is going on in this wacky code is to take a look at what PEP 380[40] has to say about using `return` with `yield`:

`return expr` in a generator causes `StopIteration(expr)` to be raised upon exit from the generator.

In this case, `StopIteration` is raised at the beginning of `my_func()` due to the `return` statement inside the function being called. Your code catches the `StopIteration` exception inside the `list()` function at the end of the code.

Because an exception is raised, `["Python"]` is not returned, so the `list()` function returns an empty list.

If you’d like to get `["Python"]` out of your code, you would need to modify the call to use the `next()` function wrapped in an exception handler:

[40] https://peps.python.org/pep-0380/

```python
def my_func(value):
    if value == 5:
        return ["Python"]
    else:
        yield from range(value)

try:
    next(my_func(5))
except StopIteration as exception:
    print(f"StopIteration caught! {exception.value = }")
```

This code removes the call to `list()`, which will automatically catch the `StopIteration` exception and uses the `next()` function instead. The `next()` function does **not** catch `StopIteration`, so you wrap that call with Python's `try / except` construct to catch that exception yourself.

To get the value of the exception, you can access the `exception` object's `value` attribute.

Quiz 55 - More Shadows

Shadows, darkness, the Will-o'-the-wisp are known to scare children. When it comes to programming, however, many developers like to write code in the dark or at the very least, engineers want to use "dark mode!"

But what happens when you name two functions the same? Is that an error? Does your code work?

This test will discover what you think you know!

The Quiz

What is the output of this silly code?

```python
def greetings():
    return "Hi there!"

def greetings(name):
    return f"Hi {name}!"

print(greetings())
```

A. Hi {name}!
B. NameError
C. TypeError
D. UnboundLocalError

Hint

You learned a little about this topic way back in **Quiz 24**!

Answer 55 - More Shadows

This is the answer to **Quiz 55 - More Shadows**.

Your original question was:

What is the output of this silly code?

```python
def greetings():
    return "Hi there!"

def greetings(name):
    return f"Hi {name}!"

print(greetings())
```

A. Hi `{name}`!
B. NameError
C. TypeError
D. UnboundLocalError

The Answer

C) TypeError

Explanation

When you run your code, you'll see that you get a `TypeError`. To be more specific, you will see something like the following:

```
builtins.TypeError: greetings() missing 1 required positional argument: 'name'
```

Python **does** allow you to create two functions with the same name. However, Python reads the functions from the top down, meaning that Python will replace the first function with the second.

The second version of the function takes an argument, so you receive a `TypeError`.

One way to fix the code would be to change how you call `greetings()`:

```python
def greetings():
    return "Hi there!"

def greetings(name):
    return f"Hi {name}!"

print(greetings("Mike"))
```

A related topic to this is called **function overloading**. You can read more about that topic in this article from Mouse vs. Python[41].

[41]https://www.blog.pythonlibrary.org/2016/02/23/python-3-function-overloading-with-singledispatch/

Quiz 56 - The Scope of the Matter

This quiz will test your knowledge of how Python's scoping rules work and what errors are raised if the scope is violated. Scope is what defines which variables are available where.

If you understand the basics, then this quiz will be a breeze!

The Quiz

What is the output when you run this code?

```python
number = 8

def adder(integer):
    print(number)
    number = 10
    print(number + integer)

adder(5)
```

A. 8 and then an exception is raised
B. NameError
C. TypeError
D. UnboundLocalError

Hint

If you get stuck, you can read up on Python's exceptions in the documentation[42].

[42] https://docs.python.org/3/library/exceptions.html

Answer 56 - The Scope of the Matter

This is the answer to **Quiz 56 - The Scope of the Matter**.

Your original question was:

What is the output when you run this code?

```python
number = 8

def adder(integer):
    print(number)
    number = 10
    print(number + integer)

adder(5)
```

A. 8 and then an exception is raised
B. NameError
C. TypeError
D. UnboundLocalError

The Answer

D) UnboundLocalError

Explanation

You get an `UnboundLocalError` because you are trying to access a variable, `number`, before it has been created in the local scope. How many scopes are there in Python? Four: Local, Enclosing, Global, and Built-in (aka LEGB).

The scope is what defines which variables are available where. Because you assigned a value to `number` in `adder`'s local scope, a new local variable `number` was created, which is different from the `number` variable in the global scope – and variable's are not allowed to change scope. So when you `print(number)` before you assign (or *bind*) `number` to `10`, Python complains about trying to use a name before you bound it.

If you remove the `number = 10` line, Python will not create a local `number` variable, so when it sees `print(number)` and `print(number + integer)`, it will fail to find `number` in `adder`'s local scope, will not find `number` in the enclosing scope (because there isn't one in this example), but will find `number` in the global scope, and use that.

As you can see, modifying global variables inside functions (or classes) can be confusing, and is usually a bad idea.

Typically, you should pass in all needed arguments to a function; something like this, for example:

```python
number = 8  # global scope

def adder(number, integer):  # local scope
    print(number)
    print(number + integer)

adder(10, 5)
```

If the value will be used often, you could instead make it an instance variable of a class:

```python
>>> class Math:
...     def __init__(self, number):
...         self.number = number
...
...     def adder(self, integer):
...         return self.number + integer
...
>>> m = Math(10)
>>> m.adder(5)
15
```

Quiz 58 - Tuple Augmented Assignment

Python supports the concept of augmented assignment. You've probably seen augmented assignment and didn't know what it was called.

Here is an example of regular assignment and an augmented assignment:

```python
# Regular assignment
number = 10

# Augmented assignment
another_number += number
```

There are multiple augmented assignment operators besides the addition variant shown above, but they all have an operator preceding the assignment operator.

This quiz will test your knowledge about tuples and augmented assignment in the REPL. Good luck!

The Quiz

What is the output of the following REPL session?

```python
>>> silly_tuple = (["one", "two"], ["three"])
>>> silly_tuple[1] += ["four"]
Traceback (most recent call last):
  Python Shell, prompt 48, line 1
builtins.TypeError: can only concatenate tuple (not "list") to tuple
>>> print(silly_tuple)
???
```

A. ([‘one’, ‘two’], [‘four’])
B. ([‘one’, ‘two’, ‘three’], [‘four’])
C. ([‘one’, ‘two’], [‘three’, ‘four’])
D. None of the above

Hint

Augmented assignment works differently than regular assignment or the `append()` method.

Answer 58 - Tuple Augmented Assignment

This is the answer to **Quiz 58 - Tuple Augmented Assignment**.

Your original question was:

What is the output of the following REPL session?

```
>>> silly_tuple = (["one", "two"], ["three"])
>>> silly_tuple[1] += ["four"]
Traceback (most recent call last):
  Python Shell, prompt 48, line 1
builtins.TypeError: can only concatenate tuple (not "list") to tuple
>>> print(silly_tuple)
???
```

A. ([‘one’, ‘two’], [‘four’])
B. ([‘one’, ‘two’, ‘three’], [‘four’])
C. ([‘one’, ‘two’], [‘three’, ‘four’])
D. None of the above

The Answer

C) ([‘one’, ‘two’], [‘three’, ‘four’])

Explanation

Wait a minute! Tuples are immutable! How can this be!?

While tuples *are* immutable (i.e., cannot change), if the tuple contains a mutable object, such as a list, that object *can* change!

Python first appends an item to the inner list in-place because `+=` invokes `__iadd__()`, the in-place method of a list object. Next, Python tries to call `__setitem__` on the tuple, which a tuple does not have. So you end up seeing a `TypeError`.

However, the error message says you *can* concatenate a tuple to a tuple using the `+=` operator: `builtins.TypeError: can only concatenate tuple (not "list") to tuple`

Let’s try that:

```
>>> silly_tuple = (["one", "two"], ["three"])
>>> another_tuple = (1, 2, 3)
>>> id(silly_tuple)
140209227484672
>>> silly_tuple += another_tuple
>>> id(silly_tuple)
140209227425808
>>> silly_tuple
(['one', 'two'], ['three'], 1, 2, 3)
```

You can use the += operator to concatenate two tuples. Notice that the tuple's identity changes (i.e., the id changes). That means that you have replaced `silly_tuple` with an entirely new tuple that contains the original's contents plus the contents of `another_tuple`.

Quiz 59 - Exception Shadowing

Exception handling is an important concept to understand. You'll be able to catch exceptions and keep your programs working longer.

Python's exception-handling machinery allows you to inspect the exception object. These objects contain useful bits of information that you might need to debug an issue with your application.

Do you know what an exception object looks like when you print it out? Let's find out!

The Quiz

What is the output of this code?

```python
e = 10

try:
    raise ZeroDivisionError
except ZeroDivisionError as e:
    pass

print(e)
```

A. 10
B. NameError: name 'e' is not defined
C. ZeroDivisionError()
D. None of the above

Hint

Look up an exception's scope and the exception object's lifespan!

Answer 59 - Exception Shadowing

This is the answer to **Quiz 59 - Exception Shadowing**.

Your original question was:

What is the output of this code?

```python
e = 10

try:
    raise ZeroDivisionError
except ZeroDivisionError as e:
    pass

print(e)
```

A. 10
B. NameError: name ‘e’ is not defined
C. ZeroDivisionError()
D. None of the above

The Answer

B) NameError: name ‘e’ is not defined

Explanation

It would be best to look at the code again to understand what is happening here.

```python
1 e = 10
2
3 try:
4     raise ZeroDivisionError
5 except ZeroDivisionError as e:
6     pass
7
8 print(e)
```

The first line of code assigns the number ten to the variable `e`. Then on the 5th line of code, you have your exception handling line where you catch the `ZeroDivisionError` exception. This line of code reassigns the `e` variable to the `ZeroDivisionError` object.

Of course, you don't do anything with the exception object in the exception block. Instead, you use the `pass` statement to ignore the exception. Once you exit the exception block, Python removes the `e` variable, which is now out of scope.

When you go to print the `e` variable, you will get a `NameError` because `e` was deleted and no longer exists.

Quiz 60 - Subclassing Silliness

Object-oriented programming (OOP) is a computer programming model that lets the engineer design around objects, which can contain data and more code.

The other primary type of programming is called functional programming. Both programming types have pros and cons, and the Python programming language excels at both types.

A subclass is where you derive from a parent class and give the subclass additional methods and attributes. For example, you might create a Ball class with a circumference and a method for rolling or bouncing. But a subclass could be a Bowling Ball, which has a specific weight and velocity.

This quiz is about subclassing a built-in type in Python and working with dictionaries. Do your best!

The Quiz

What are the contents of the `silly_dict` variable?

```python
class MyClass(str):
    pass

silly_dict = {"py": "amazing"}
p = MyClass("py")
silly_dict[p] = 10
print(silly_dict)
```

A. `{"py": "amazing"}`
B. `{"py": 10}`
C. `{"py": "amazing", <__main__.MyClass object at 0x7fda59c38460>: 10}`
D. None of the above

Hint

Test what is in the variable `p`, and you might be able to figure this one out.

Answer 60 - Subclassing Silliness

This is the answer to **Quiz 60 - Subclassing Silliness**.

Your original question was:

What are the contents of the `silly_dict` variable?

```python
class MyClass(str):
    pass

silly_dict = {"py": "amazing"}
p = MyClass("py")
silly_dict[p] = 10
print(silly_dict)
```

A. `{"py": "amazing"}`
B. `{"py": 10}`
C. `{"py": "amazing", <__main__.MyClass object at 0x7fda59c38460>: 10}`
D. None of the above

The Answer

B) `{"py": 10}`

Explanation

This quiz is interesting. Here you subclass Python's built-in `str` class, but you don't add any new methods or attributes. In effect, you have created a new object that behaves the same as a regular Python string object.

What does that mean, though? Look at the following code to get some illumination:

```
>>> silly_dict = {"py": "amazing"}
>>> p = "py"
>>> silly_dict[p] = 10
>>> silly_dict
{'py': 10}
>>> silly_dict["py"] = "amazing"
>>> silly_dict
{'py': 'amazing'}
>>> class MyClass(str):
...     pass
...
>>> p = MyClass("py")
>>> p == "py"
True
>>> silly_dict[p] = "PHP"
>>> silly_dict
{'py': 'PHP'}
```

You can change a value in a dictionary by setting the dictionary's key to something else.

Often, you do that by setting the key explicitly, like this:

```
>>> silly_dict["py"] = "amazing"
```

But you can also save the key to a variable and then set the dictionary key's value using the variable name:

```
>>> p = "py"
>>> silly_dict[p] = 10
```

You created a new string type when you subclassed the `str` class. So when you made the `MyClass("py")` object and assigned it to `p`, you could then use that string-like object to set the dictionary key to a new value:

```
>>> class MyClass(str):
...     pass
...
>>> p = MyClass("py")
>>> silly_dict[p] = "PHP"
>>> silly_dict
{'py': 'PHP'}
```

Pretty neat!

Quiz 61 - Loopy Variables

The scope of a variable can be a confusing thing. You've seen how scope can affect variables in functions and exception handlers in recent quizzes.

But there are other types of scope. You'll be looking at a new one here… or will you?

C++ handles the scope of a `for` loop's variable differently than Python does. Do you know what Python does, though?

Let's find out!

The Quiz

What is the output of this Python REPL session?

```
>>> number = "zero"
>>> for number in range(10):
...     print(number * 2)
...
0
2
4
6
8
10
12
14
16
18
>>> print(number)
```

A. 'zero'
B. 18
C. 9
D. Something else

Hint

Remember your lessons from previous quizzes about shadowing. They may help you here.

Answer 61 - Loopy Variables

This is the answer to **Quiz 61 - Loopy Variables**.

Your original question was:

What is the output of this Python REPL session?

```
>>> number = "zero"
>>> for number in range(10):
...     print(number * 2)
...
0
2
4
6
8
10
12
14
16
18
>>> print(number)
```

A. 'zero'
B. 18
C. 9
D. Something else

The Answer

C) 9

Explanation

When you create a `for` loop in Python, you must create a variable to hold an item from the object you are iterating over. That variable still exists when the `for` loop ends, and some authors call this a leaky variable.

Other languages, such as C++, don't allow this sort of thing to happen. Instead, that variable would only exist within the `for` loop code block in its own scope. In Python, though, the `number` before the `for` loop and the `number` in the `for` loop are in the same scope and the same variable.

For the code in this quiz, `number` will change on each iteration as it loops over a range of zero to nine. The number nine is the last integer that `number` will be set to, which is why **C** is the correct answer.

Quiz 62 - More Loopy Variables

In the last quiz, you learned that Python `for` loops do not create their own scope. But have you ever tried to access the variable used in a list comprehension?

Hopefully, you know the answer to that question, as that is what this quiz is about!

Take a look.

The Quiz

What is the value of the `number` variable at the end of this code?

```python
number = 5
lots_of_numbers = [number for number in range(5, 50, 2)]
print(number)
```

A. 5
B. 49
C. 50
D. None of the above

Hint

Try it in a REPL!

Answer 62 - More Loopy Variables

This is the answer to **Quiz 62 - More Loopy Variables**.

Your original question was:

What is the value of the `number` variable at the end of this code?

```python
number = 5
lots_of_numbers = [number for number in range(5, 50, 2)]
print(number)
```

A. 5
B. 49
C. 50
D. None of the above

The Answer

A) 5

Explanation

List comprehensions in Python 3 do **not** leak their `for` loop variables, as a new scope is created to run the comprehension.

However, if you were to run this code in Python 2, it DOES leak!

So, the answer in Python 3 is **A**, but if you run it in Python 2, the answer would be **B**.

But why would you run it in Python 2? Don't do that!

Quiz 63 - Dictionary Madness

Dictionaries are handy data types in Python. They give you a super-fast lookup and can hold any object as their value.

This quiz tests your knowledge of how to create a dictionary. Does the weird syntax work? If it does, what does the dictionary contain?

Let's find out!

The Quiz

What is the output of this crazy code?

```python
x, y = x[y] = {}, 10
print(x)
```

A. SyntaxError
B. `{}`
C. `{10: (10, 10)}`
D. `{10: ({...}, 10)}`
E. What the heck?!

Hint

Do you know what recursion is?

Answer 63 - Dictionary Madness

This is the answer to **Quiz 63 - Dictionary Madness**.

Your original question was:

What is the output of this crazy code?

```
x, y = x[y] = {}, 10
print(x)
```

A. SyntaxError
B. `{}`
C. `{10: (10, 10)}`
D. `{10: ({...}, 10)}`
E. What the heck?!

The Answer

D) `{10: ({...}, 10)}`

Explanation

While **D** is the correct answer, **E** wouldn't have been a bad choice. What the heck is going on here?

This code is recursively constructing a dictionary of dictionaries that contain the same thing.

Here's a REPL session that shows what's going on:

```
>>> x, y = x[y] = {}, 10
>>> x
{10: ({...}, 10)}
>>> type(x[10])
<class 'tuple'>
>>> type(x[10][0])
<class 'dict'>
>>> x[10][0]
{10: ({...}, 10)}
>>> x[10][0][10]
```

```
({10: (...)}, 10)
>>> x[10][0][10][0]
{10: ({...}, 10)}
>>> x[10][0][10][0][10]
({10: (...)}, 10)
>>> x[10][0][10][0][10][0]
{10: ({...}, 10)}
```

As you can see, `x` contains a dictionary with a single item. The key is `10`, and the value is a tuple: (, 10)

If you drill down into the tuple, you will find that it too contains a dictionary with a single key: value pair that is identical to the upper level dictionary. You can drill down again and keep finding that it is dictionaries all the way down.

Should you write code like this? Absolutely not. This is an anti-pattern at best. But it's a neat party trick and may give you something to discuss at a code meetup or conference.

Quiz 64 - Silly Lists

Lists are another handy data type in the Python programming language. Other programming languages call them arrays.

One of the many nice things about Python lists is that they can hold any data type, including other lists.

This quiz will test your knowledge of lists and equality!

The Quiz

What is the output of this silly code?

```python
some_list = some_list[0] = [0]
print(some_list[0][0][0][0][0][0] == some_list)
```

A. True
B. False
C. An exception is thrown

Hint

Always keep in mind that identity is not the same as equality.

Answer 64 - Silly Lists

This is the answer to **Quiz 64 - Silly Lists**.

Your original question was:

What is the output of this silly code?

```python
some_list = some_list[0] = [0]
print(some_list[0][0][0][0][0][0] == some_list)
```

A. True
B. False
C. An exception is thrown

The Answer

A) True

Explanation

The strange syntax you see here creates a similar object to the previous quiz, except that this time you end up with a nested list instead of a nested dictionary.

If you print out this list, you'll see something odd:

```
>>> some_list
[[...]]
```

Once again, you see the pesky ellipsis. That ellipsis tells you that you have another infinite object on your hands. In this case, it's an infinity list!

Here is the proof:

```
>>> len(some_list)
1
>>> len(some_list[0])
1
>>> len(some_list[0][0])
1
>>> some_list[0][0][0][0][0][0]
[[...]]
>>> some_list
[[...]]
```

The length of the infinity list is always one, and the one item is itself a list with another list inside of it.

When you print out `some_list` and `some_list[0][0][0][0][0][0]`, you can see that the output is the same.

That means that when you test if they are equal to each other, Python returns `True`.

Do not write code like this! While it's fun and entertaining, it's also hard to understand and not especially useful in your regular day-to-day programming.

Quiz 65 - Modifying Lists

Looping over lists, dictionaries and other iterables is a routine activity when you're a computer programmer. One of the items you need to keep in mind when iterating over an iterable is whether or not to load the entire thing into memory.

In fact, that is a common interview question. Should you load a giant file into memory and process it? Usually, the answer is no!

But enough about memory concerns and stack overflows. What you came here to do was a quiz! What happens when you iterate over something and modify the thing that you are iterating over?

The Quiz

What is the output of this code?

```python
animals = {1: "Python"}
loops = 1
for key in animals:
    del animals[key]
    animals[key + 1] = None
    print(f"{loops = }")
    loops += 1
```

A. loops = 1
B. Answer A plus: loops = 2
C. Answer A and then an exception is thrown
D. None of the above

Hint

You'll have to play around with this one in the REPL to figure it out!

Answer 65 - Modifying Lists

This is the answer to **Quiz 65 - Modifying Lists**.

Your original question was:

What is the output of this code?

```python
animals = {1: "Python"}
loops = 1
for key in animals:
    del animals[key]
    animals[key + 1] = None
    print(f"{loops = }")
    loops += 1
```

A. loops = 1
B. Answer A plus: loops = 2
C. Answer A and then an exception is thrown
D. None of the above

The Answer

C) Answer A and then an exception is thrown

Explanation

Python will delete the key from the dictionary when you run this code. Then Python will add a new key, value pair: `2: None`

The code then prints out which loop you are on and increments the `loops` variable. Finally, Python tries to reaccess the dictionary to see if it has more keys.

But wait! While iterating over your dictionary, the dictionary changed! Python doesn't like that, so it raises an exception and lets you know you did a bad thing: `RuntimeError: dictionary keys changed during iteration`

The lesson here is that you should **not** add or remove keys in a dictionary that you are currently iterating over. Changing values is fine, though.

Quiz 66 - Garbage Collection

Python has a garbage collector that helps Python manage its memory usage. Python uses an algorithm that uses reference counting. A reference is one or more variables pointing at the same value.

This quiz will test your knowledge of using classes to track when a reference is deleted.

Good luck!

The Quiz

What is the output of the following REPL session, if any?

```
>>> class Spam:
...
...     def __del__(self):
...         print("Deleted!")
...
>>> x = Spam()
>>> y = x
>>> del x
```

A. Deleted!
B. Deleted! (gets printed twice)
C. Nothing is printed
D. None of the above

Hint

You can learn more about garbage collection in Python in the documentation[43].

[43]https://devguide.python.org/internals/garbage-collector/

Answer 66 - Garbage Collection

This is the answer to **Quiz 66 - Garbage Collection**.

Your original question was:

What is the output of the following REPL session, if any?

```
>>> class Spam:
...
...     def __del__(self):
...         print("Deleted!")
...
>>> x = Spam()
>>> y = x
>>> del x
```

A. Deleted!
B. Deleted! (gets printed twice)
C. Nothing is printed
D. None of the above

The Answer

C) Nothing is printed

Explanation

How does the garbage collector work in Python?

Here is an example from the Garbage Collection Design document[44] in the Python dev guide:

[44]https://devguide.python.org/internals/garbage-collector/

```
>>> x = object()
>>> sys.getrefcount(x)
2
>>> y = x
>>> sys.getrefcount(x)
3
>>> del y
>>> sys.getrefcount(x)
2
```

When a value is assigned to a variable, the reference count is incremented. Here you have `x` with a reference count of two. When you assign `x` to `y`, you increase x's reference count by one.

The same thing applies with the `Spam()` object reference count in your quiz:

```
>>> class Spam:
...
...     def __del__(self):
...         print("Deleted!")
...
>>> x = Spam()
>>> y = x
>>> del x
```

Here you create two references to the same `Spam()` object. Then you delete the original reference, but the `Spam()` object continues to live because `y` still references it. That means nothing is printed yet because a reference still exists.

The `__del__()` method is also known as a **destructor** method. The `__del__()` method is called when all references to the object are deleted. If you want to see something printed, you will need to delete `y` too.

Note: If you save this code to a Python file and run it, you will see it print "Deleted!". That is because the `y` variable gets deleted by Python when it reaches the end of your code file when you run it. When `y` goes away, it causes `__del__()` to be invoked.

Give it a try and experiment some more to see if you can learn anything else about Python garbage collection!

Quiz 67 - Details Matter

Reading and understanding other people's code is a skill you must hone to succeed. Reading comprehension will help you learn new code more quickly, and you will also be better able to find and correct mistakes.

Code errors don't always raise exceptions, and sometimes they are logic issues. Other times, errors occur because the developer doesn't understand how a language works.

Today you'll need to figure out if there's something wrong with this code. And if there is, what the output will be!

The Quiz

What is the output when you run this code?

```python
my_list = ["uno",
           "dos"
           "tres",
           "quatro"
           "cinco"]

print(len(my_list))
```

A. 5
B. 4
C. 3
D. 2
E. An exception is raised

Hint

Quiz 50 might give you a clue into what's happening here.

Answer 67 - Details Matter

This is the answer to **Quiz 67 - Details Matter**.

Your original question was:

What is the output when you run this code?

```
my_list = ["uno",
           "dos"
           "tres",
           "quatro"
           "cinco"]

print(len(my_list))
```

A. 5
B. 4
C. 3
D. 2
E. An exception is raised

The Answer

C) 3

Explanation

The quickest way to understand what is happening here is to simply print the list out:

```
>>> print(my_list)
['uno', 'dostres', 'quatrocinco']
```

When you have a list of strings in Python, the strings must have commas to separate them. If you don't include a comma, then the strings are combined. The first item in the list has a comma, so it is one item. The second does not, so it is combined with the third item which DOES have a comma.

The fourth item also doesn't have a comma, so it is combined with the fifth item. That results in a list of three strings!

Quiz 68 - Range Rules

Python includes over 50 built-in functions and classes you can use for many different things.

One of the most popular is the `range()` "function". If you look in the documentation[45], you'll find that `range()` is an immutable sequence rather than a function.

You can use `range()` to quickly create a Python list using the `list()` function or in a list comprehension.

Today you will be tested on your knowledge of what parameters `range()` accepts.

The Quiz

What is the output of the following code?

```python
print([number for number in range(25, 35, 2)])
```

A. builtins.TypeError: range expected at most 2 arguments, got 3
B. [25, 26, 29, 30, 31, 32, 33, 34]
C. [25, 27, 29, 31, 33, 35]
D. [25, 27, 29, 31, 33]

Hint

Find out what the **step** parameter is.

[45]https://docs.python.org/3/library/stdtypes.html#typesseq-range

Answer 68 - Range Rules

This is the answer to **Quiz 68 - Range Rules**.

Your original question was:

What is the output of the following code?

```python
print([number for number in range(25, 35, 2)])
```

A. builtins.TypeError: range expected at most 2 arguments, got 3
B. [25, 26, 29, 30, 31, 32, 33, 34]
C. [25, 27, 29, 31, 33, 35]
D. [25, 27, 29, 31, 33]

The Answer

D) [25, 27, 29, 31, 33]

Explanation

According to the Python documentation[46], the `range()` type accepts up to three parameters:

- start
- stop
- step

The `start` argument tells `range()` where to start, while the `stop` argument is the end of the sequence. Take note that the end value of the sequence is not included in the sequence.

When you use a positive `step` argument, the documentation says that the contents of the range "are determined by the formula `r[i] = start + step*i` where `i >= 0` and `r[i] < stop`."

In the code above, you start at twenty-five with a step of two and go until you reach thirty-five, but you don't include thirty-five in the sequence. This gives you `[25, 27, 29, 31, 33]`.

[46]https://docs.python.org/3/library/stdtypes.html#ranges

Quiz 69 - Integer Identity

Python lets you check the identity of two objects using the `is` keyword. Most of the time, you will only care if two objects are equal rather than if they are the same object.

However, this quiz aims to test your knowledge of Python and how it handles identity for integers.

Good luck!

The Quiz

Take a look at the following CPython REPL session:

```
>>> a = 1
>>> b = 1
>>> a is b
True
```

Which value will no longer return `True` if you set both variables to that value?

A. 10
B. 200
C. 256
D. 257

Hint

If you get stuck, you can look up the following term: integer interning.

Answer 69 - Integer Identity

This is the answer to **Quiz 69 - Integer Identity**.

Your original question was:

Take a look at the following CPython REPL session:

```
>>> a = 1
>>> b = 1
>>> a is b
True
```

Which value will no longer return `True` if you set both variables to that value?

A. 10
B. 200
C. 256
D. 257

The Answer

D) 257

Explanation

Did you know there are different distributions of Python? The one you get from the Python website[47] is called CPython. But there are others, such as PyPy.

This is important because they each have their own optimizations and slight differences. In this book, you will focus on CPython. CPython has an optimization that is called **integer interning**. When Python starts, it pre-loads (or caches) a list of integers in the range of -5 to 256.

Interning means that values in that range have the same identity and value (or equality). But values outside of that range do not have the same identity.

Here is a REPL session that illustrates the concept:

[47]https://www.python.org/

```
>>> a = 1
>>> b = 1
>>> a is b
True
>>> a = 256
>>> b = 256
>>> a is b
True
>>> a = 257
>>> b = 257
>>> a is b
False
```

You can read more about this topic in the following article: Python 101: Equality vs. Identity[48].

[48]https://www.blog.pythonlibrary.org/2017/02/28/python-101-equality-vs-identity/

Quiz 70 - List Insanity

It is time for another quiz about Python lists! The list is a great way to hold a series of objects. You can create a list of strings or integers or other objects, and you can also create a list that contains multiple different types of objects!

A Python list has multiple methods you can use to work with the list.

This quiz will test what you know about how lists work!

The Quiz

What is the output of this REPL session?

```
>>> snakes = ["Python", "Garter", "Anaconda"]
>>> def add_snake(snake_type):
...     snakes.extend(snake_type)
...     print(snakes)
...
>>> add_snake("Boa")
```

A. ["Python", "Garter", "Anaconda"]
B. ["Python", "Garter", "Anaconda", "Boa"]
C. builtins.NameError: name 'snakes' is not defined
D. ['Python', 'Garter', 'Anaconda', 'B', 'o', 'a']
E. UnboundLocalError

Hint

Check out Python's list methods to see how they work, or try out `extend()` in the REPL.

Answer 70 - List Insanity

This is the answer to **Quiz 70 - List Insanity**.

Your original question was:

What is the output of this REPL session?

```
>>> snakes = ["Python", "Garter", "Anaconda"]
>>> def add_snake(snake_type):
...     snakes.extend(snake_type)
...     print(snakes)
...
>>> add_snake("Boa")
```

A. [“Python”, “Garter”, “Anaconda”]
B. [“Python”, “Garter”, “Anaconda”, “Boa”]
C. builtins.NameError: name ‘snakes’ is not defined
D. [‘Python’, ‘Garter’, ‘Anaconda’, ‘B’, ‘o’, ‘a’]
E. UnboundLocalError

The Answer

D) [‘Python’, ‘Garter’, ‘Anaconda’, ‘B’, ‘o’, ‘a’]

Explanation

The `snakes` variable is global in this context. Because it is global, E) cannot be the answer.

But what happens when you pass a string to a list’s `extend()` method? The `extend()` method takes in an iterable. A string is iterable in Python.

So when you pass a string to `extend()`, it iterates over the characters in the string and appends each character to the list.

While `extend()`’s behavior is handy when you want to add one list’s contents to another list, you usually don’t want to add individual letters from a string to a list. In this case, the `append()` method would have made more sense:

```
>>> snakes = ["Python", "Garter", "Anaconda"]
>>> def add_snake(snake_type):
...     snakes.append(snake_type)
...     print(snakes)
>>> add_snake("Boa")
["Python", "Garter", "Anaconda", "Boa"]
```

If you had used `append()` instead of `extend()`, then the answer would have been **B)**.

Quiz 71 - Enumerate Enigma

Python has dozens of handy built-in functions. This quiz will focus on what `enumerate()` does.

As you may know, `enumerate()` is an excellent way to iterate over the index and items in an iterable, such as a list.

Let's see if you can figure this puzzler out!

The Quiz

What is the output of the following code?

```python
my_list = [1, 2, 3, 4]
for index, item in enumerate(my_list):
    del item

print(my_list)
```

A. Nothing. An exception is raised
B. []
C. [1, 2, 3, 4]
D. None of the above

Hint

If you get stuck, you might want to look up what `enumerate()` returns.

Answer 71 - Enumerate Enigma

This is the answer to **Quiz 71 - Enumerate Enigma**.

Your original question was:

What is the output of the following code?

```python
my_list = [1, 2, 3, 4]
for index, item in enumerate(my_list):
    del item

print(my_list)
```

A. Nothing. An exception is raised
B. []
C. [1, 2, 3, 4]
D. None of the above

The Answer

C) [1, 2, 3, 4]

Explanation

The key here is that `enumerate()` returns indexes and items. When you tell Python to delete an item in the code above, you are **not** telling it to delete the item from the list!

Instead, you are deleting the `item` variable. If you want to remove an item from the list, you should use a list method, such as `pop()` or `remove()`.

But you didn't, so your list is completely unaffected by your deletes!

Quiz 72 - Walrus Comprehensions

You have seen several quizzes where you learned about comprehensions in Python. A comprehension is a one-line `for` loop that you can use to create a list, dictionary, or generator.

However, you will throw a wrench into the works to make things more interesting by adding an assignment expression. The assignment expression (AKA the walrus operator) allows you to assign values to variables inside an expression.

Let's see if you understand what is happening in this quiz!

The Quiz

What is the value of `number` at the end of this code?

```python
number = 10
lots_of_numbers = [number for x in
                   range(5, 100, 2)
                   if (number := x)]
print(number)
```

A. 5
B. 19
C. 100
D. 99

Hint

You looked into loops and variable scope back in **Quiz 61** and **62**. They may help.

Answer 72 - Walrus Comprehensions

This is the answer to **Quiz 72 - Walrus Comprehensions**.

Your original question was:

What is the value of `number` at the end of this code?

```python
number = 10
lots_of_numbers = [number for x in
                   range(5, 100, 2)
                   if (number := x)]
print(number)
```

A. 5
B. 19
C. 100
D. 99

The Answer

D) 99

Explanation

You learned about Assignment Expressions way back in **Quiz 21**. The assignment expression lets you assign values inside an expression, starting in Python 3.8[49]. You can learn more about them in PEP 572[50].

Normally, a list comprehension doesn't leak a variable. You can see that by removing the assignment expression like this:

[49] https://docs.python.org/3/whatsnew/3.8.html#assignment-expressions
[50] https://peps.python.org/pep-0572/

```
>>> number = 10
>>> lots_of_numbers = [number for x in
...                     range(5, 100, 2)]
...
>>> number
10
```

The variable, `number`, remains unchanged even though the `number` variable changes quite a bit inside of the list comprehension. This happens because the `number` variable inside the list comprehension is in a different scope and does **not** overwrite the `number` variable outside of the list comprehension.

When you throw the assignment expression into the mix, that changes things though. Now you are reassigning `x` to `number` in every iteration inside the list comprehension.

The lesson here is that you are making wacky code and doing something you shouldn't do. This code should fail a peer review because it's hard to understand and deliberately forces `number` to be changed inside of the list comprehension.

Quiz 73 - Indexing Indignation

You may recall that Python's `range()` function is built-in. In fact, `range()` isn't so much a function as it is an "an immutable sequence of numbers".

You can use `range()` to return a sequence of numbers and easily create other data types.

In this quiz, you'll need to study the code and determine whether this is a syntax error. If it's not, then what will the output be? Good luck!

The Quiz

What is the output of this odd code?

```python
class Number:
    integer = 0
    def __index__(self):
        internal_int = Number.integer
        Number.integer += 10
        return internal_int

print([x for x in range(Number(), Number())])
```

A. An exception is raised
B. [1, 2, 3, 4, 5]
C. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
D. []

Hint

This quiz came from reading the Python documentation[51].

[51] https://docs.python.org/3/library/stdtypes.html#typesseq-range

Answer 73 - Indexing Indignation

This is the answer to **Quiz 73 - Indexing Indignation**.

Your original question was:

What is the output of this odd code?

```python
class Number:
    integer = 0
    def __index__(self):
        internal_int = Number.integer
        Number.integer += 10
        return internal_int

print([x for x in range(Number(), Number())])
```

A. An exception is raised
B. [1, 2, 3, 4, 5]
C. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
D. []

The Answer

C) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Explanation

Python's documentation[52] contains the answer to this screwy quiz:

"The arguments to the range constructor must be integers (either built-in int or any object that implements the **index**() special method)."

So, it turns out that you can create a class that implements the `__index__` magic method and use that to create a range of integers. How interesting!

Let's look at that quiz code again:

[52]https://docs.python.org/3/library/stdtypes.html#typesseq-range

```python
class Number:
    integer = 0
    def __index__(self):
        internal_int = Number.integer
        Number.integer += 10
        return internal_int

print([x for x in range(Number(), Number())])
```

You create a class attribute named `integer` and set it to zero. Then you implement the `__index__()` method. The method stores the class attribute's current value in `internal_int`, then it promptly increments that value by ten. Finally, you return the original value.

The first time you create a `Number()` object inside your `range()`, the `range()` object looks for an integer or an object with `__index__()` implemented. `Number()` has an `__index__()` method, which returns zero the first time you create a `Number()` object.

Each time you create a new `Number()` object, the value returned has been incremented by ten.

Check it out in this REPL session:

```python
>>> class Number:
...     integer = 0
...     def __index__(self):
...         internal_int = Number.integer
...         Number.integer += 10
...         return internal_int
...
>>> Number().__index__()
0
>>> Number().__index__()
10
>>> Number().__index__()
20
>>> Number().__index__()
30
```

Because of this behavior, you are creating a range from zero to ten for the quiz, but not including ten.

Quiz 74 - List Defaults

Python functions have several different types of arguments. The two most common arguments are **positional** and **keyword** arguments. Another popular type of argument is the default argument.

If your function has one or more arguments that are set to a default value, when you invoke that function, you don't need to specify anything for that value unless you want it to be different than the default.

Your quiz will test your knowledge of default arguments in functions. Have fun!

The Quiz

What is the output of this code?

```python
def my_function(default=[]):
    default.append("Python")
    return default

my_function()
print(my_function())
```

A. []
B. ['Python', 'Python']
C. ['Python']
D. None of the above

Hint

Mutable objects as defaults can be tricky.

Answer 74 - List Defaults

This is the answer to **Quiz 74 - List Defaults**.

Your original question was:

What is the output of this code?

```python
def my_function(default=[]):
    default.append("Python")
    return default

my_function()
print(my_function())
```

A. []
B. [‘Python’, ‘Python’]
C. [‘Python’]
D. None of the above

The Answer

B) [‘Python’, ‘Python’]

Explanation

When you use a mutable data type as a default argument in a function, you are asking for trouble. The mutable object will “remember” what’s in it every time you call it unless you pass in a new object.

You can run into weird issues with unexpected data when you use a mutable object, like a dictionary or a list, as a default argument. You’ll find yourself scratching your head and wondering how that got in there!

Look at the following examples to see how using a Python list as a default argument can cause some weird issues:

```
>>> def my_function(default=[]):
...     default.append("Python")
...     return default
...
>>> my_function()
['Python']
>>> my_function()
['Python', 'Python']
>>> my_function()
['Python', 'Python', 'Python']
```

Here's another one that brings home how confusing this can be:

```
>>> def my_function(default=[]):
...     default.append("Python")
...     return default
...
>>> my_function(["snake"])
['snake', 'Python']
>>> my_function()
['Python']
```

You might have expected that last call to `my_function()` to return `['snake', 'Python', 'Python']`, but it didn't. That's because you weren't using the `default` list in the first call. When you call `my_function()` the second time, it *does* use the `default` list, which is initially empty, and the function appends "Python" to it.

So is there ever a good place to use mutable defaults? Yes! Mutable defaults can be very useful for caching and/or recursive algorithms:

```
def fibonacci(n, cache={0:0, 1:1}):
    if n in cache:
        return cache[n]
    else:
        value = fibonacci(n-1) + fibonacci(n-2)
        cache[n] = value
        return value
```

The moral of this story is: **Excercise extreme care when using mutable objects as default values in a function or method.**

Quiz 75 - Counting Letters

Many companies like to give interview questions similar to this quiz. The idea is to have the potential new employee show off their skills in solving an obtuse problem programmatically.

You'll also see problems similar to this one on code challenge websites. Learning how to find the answer to this question can be quite helpful in getting better at the Python programming language.

Give it a try and see how you do!

The Quiz

Find the top 3 letter counts in the following string using Python:

```
words = "The little red fox jumped over a fence and bit a python"
```

Good luck!

Hint

The `collections` module has one way to do it.

Answer 75 - Counting Letters

This is the answer to **Quiz 75 - Counting Letters**.

Your original question was:

Find the top 3 letter counts in the following string using Python:

```python
words = "The little red fox jumped over a fence and bit a python"
```

The Answer

There are multiple ways to find the answer to this question. One of the simplest ways is to use the `collections` module.

Let's find out how!

Explanation

Before you learn the "easy" way to solve this quiz, let's look at how to solve it using regular Python code:

```python
def solution(letters):
    letters = letters.lower()
    my_dict = {}
    for letter in letters:
        if letter in my_dict:
            my_dict[letter] += 1
        else:
            my_dict[letter] = 1
    sorted_by_value = sorted(my_dict.items(), key=lambda item: item[1])
    return sorted_by_value[-3:]
```

The first step is to make all the letters lowercase unless the quiz specifically asks you to keep track of both lower and upper case letters. This one doesn't, so lowering them makes sense.

The next step is to create a dictionary. Then you'll loop over each character in your string and check if the letter is in your dictionary. If it isn't, you will add it with a count of one. If the letter is in the dictionary, you'll increment its count.

Since you want to get the top three letter counts, you'll need to sort the data. You use Python's `sorted()` function to sort the dictionary by value. The sorted function takes in an iterable, a list-like set of keys and values returned from the dictionary's `items()` method. Then you use a lambda to sort by the second item in the dictionary, which is the value.

Now you have a sorted list, from lowest letter count to highest. You want the three highest counts, which is as simple as grabbing the last three items. The result will look like this: `[('t', 5), ('e', 7), (' ', 11)]`

The most common character is the space character!

You are probably wondering how you might do this with the `collections` module. Take a look:

```python
import collections

def solution(letters):
    letters = letters.lower()
    count = collections.Counter(letters)
    return count.most_common(3)
```

Here you only need three lines of code. The first line puts all the characters in lowercase. Then you create an instance of `collections.Counter()`, which will count your iterable and return a dict-like object.

The `Counter` object has a handy method called `most_common()`, which you can use to get a sorted list of items from highest to lowest. You can also specify how many of those counts you want. You want the top three in your case, so you pass three to `most_common()`.

Here is the result: `[(' ', 11), ('e', 7), ('t', 5)]`

As you can see, the result is very similar, although the three letter counts are reversed from highest to lowest in this case.

If you have the time, you should read about the collections module[53], as it has several other great tools in it that you may find useful!

[53]https://docs.python.org/3/library/collections.html

Quiz 76 - Veritable Variables

Python 3.6 added the concept of **variable annotation** back in 2016. Originally, you could type hint arguments in functions and methods, but if you wanted to add a type hint to a variable, you were out of luck.

You can read about variable annotation in PEP 526[54]. But what does all this have to do with your quiz?

That is what you'll need to figure out!

The Quiz

What is the output of this code?

```
def area(width, height):
    print(f"{width = } {height = }")
    return width * height

if __name__ == "__main__":
    width: int
    height = 15
    area(width, height)
```

A. SyntaxError
B. NameError
C. width = None height = 15
D. width = 15 height = None

Hint

Does variable annotation require you to initialize the variable?

[54]https://peps.python.org/pep-0526/

Answer 76 - Veritable Variables

This is the answer to **Quiz 76 - Veritable Variables**.

Your original question was:

What is the output of this code?

```python
def area(width, height):
    print(f"{width = } {height = }")
    return width * height

if __name__ == "__main__":
    width: int
    height = 15
    area(width, height)
```

A. SyntaxError
B. NameError
C. width = None height = 15
D. width = 15 height = None

The Answer

B) NameError

Explanation

When you run the quiz code, you'll find that Python raises the following: `builtins.NameError: name 'width' is not defined`.

This exception occurs on the very last line of the code. You might think that this line would be the problem:

```python
width: int
```

However, that is valid Python code. You can't use `width` without initializing it to something. Here's a simple fix:

```python
def area(width, height):
    print(f"{width = } {height =}")
    return width * height

if __name__ == "__main__":
    width: int
    height = 15
    width = 10
    area(width, height)
```

Here you add a variable annotation, and then before you call `area()`, you set `width` to a value. Alternatively, you could change the `width` code to the following:

```python
width: int = 10
```

In this example, you annotate AND initialize the `width` variable.

Quiz 77 - Tuples Are Immutable!

It’s back to basics with this quiz. Today, you will test your mettle with a puzzle on immutability vs. mutability.

Which data types in Python are mutable? Which are not? Can they be mixed?

If you know the answers to those questions, you’ll figure this one out quickly!

The Quiz

What is the output of this silly code?

```python
my_tuple = ([1, 2], ["three", "four"])
my_tuple[1].append("five")
print(my_tuple)
```

A. AttributeError
B. TypeError
C. ([1, 2, ‘five’], [‘three’, ‘four’])
D. ([1, 2], [‘three’, ‘four’, ‘five’])

Hint

Read the code carefully!

Answer 77 - Tuples Are Immutable!

This is the answer to **Quiz 77 - Tuples Are Immutable!**.

Your original question was:

What is the output of this silly code?

```
my_tuple = ([1, 2], ["three", "four"])
my_tuple[1].append("five")
print(my_tuple)
```

A. AttributeError
B. TypeError
C. ([1, 2, ‘five’], [‘three’, ‘four’])
D. ([1, 2], [‘three’, ‘four’, ‘five’])

The Answer

D) ([1, 2], [‘three’, ‘four’, ‘five’])

Explanation

While tuples are certainly immutable, that doesn’t mean that a tuple cannot contain a mutable object.

Therein lays the key to this quiz. Because tuples can contain lists, dictionaries, and other mutable types, you *can* change those inner objects while **not** changing the tuple’s identity!

Take a look:

```
>>> my_tuple = ([1, 2], ["three", "four"])
>>> id(my_tuple)
140396831129856
>>> my_tuple[1].append("five")
>>> id(my_tuple)
140396831129856
```

Here you can see that changing the contents of a list inside a tuple doesn’t change the tuple’s identity. What that means is that the tuple isn’t being replaced. The main thing to understand here is that you are not changing the tuple itself. You are changing the mutable object inside of it.

Quiz 78 - Tuple Addition

Tuples are immutable, while lists and dictionaries are not. Did you know that because a tuple is immutable, you can use it as a key in a dictionary? Now you do!

Last time you learned about what happens when you modify a tuple that contains a list. In this quiz, you'll try using augmented assignment.

What happens when you use += with a tuple? Good luck!

The Quiz

What happens when you run this code?

```python
my_tuple = ("Python", "cobra")
my_tuple += ("boa", "anaconda")
print(my_tuple)
```

A. TypeError
B. ('Python', 'cobra', 'boa', 'anaconda')
C. SyntaxError
D. ('Python', 'cobraboa', 'anaconda')

Hint

Do tuples support augmented assignment?

Answer 78 - Tuple Addition

This is the answer to **Quiz 78 - Tuple Addition**.

Your original question was:

What happens when you run this code?

```python
my_tuple = ("Python", "cobra")
my_tuple += ("boa", "anaconda")
print(my_tuple)
```

A. TypeError
B. ('Python', 'cobra', 'boa', 'anaconda')
C. SyntaxError
D. ('Python', 'cobraboa', 'anaconda')

The Answer

B) ('Python', 'cobra', 'boa', 'anaconda')

Explanation

A tuple is immutable, but you can use augmented assignment to create a new tuple!

Yes, what you are doing in this quiz's code is replacing the original tuple with a brand-new one that contains four strings.

You can see that if you print out the `id` of the tuple over time:

```python
>>> my_tuple = ("Python", "cobra")
>>> id(my_tuple)
140396830117568
>>> my_tuple += ("boa", "anaconda")
>>> id(my_tuple)
140396834568768
```

You will get different ids on your machine when you run this code. But you'll notice that the identity of `my_tuple` changes after adding the two strings. That means that `my_tuple` is replaced with a new tuple!

Quiz 79 - Exacerbated Exceptions

Learning how to catch exceptions in Python is an important skill. You want your code to be robust and not crash when the program encounters bad data.

Fortunately, Python has the `try / except / finally` construct, which makes building exception handling nice.

There are multiple ways to catch more than one exception type. This quiz will test your knowledge of how exception handling works.

Have fun!

The Quiz

What happens when you run this code?

```
my_list = ["Python", "Boa", "Anaconda"]
try:
    print(my_list[4])
except IndexError, ValueError:
    print("Caught an error!")
```

A. ‘Caught an error!’
B. [“Python”, “Boa”, “Anaconda”]
C. SyntaxError
D. None of the above

Hint

You need to understand how to write exception handlers in Python.

Answer 79 - Exacerbated Exceptions

This is the answer to **Quiz 79 - Exacerbated Exceptions**.

Your original question was:

What happens when you run this code?

```python
my_list = ["Python", "Boa", "Anaconda"]
try:
    print(my_list[4])
except IndexError, ValueError:
    print("Caught an error!")
```

A. 'Caught an error!'
B. ["Python", "Boa", "Anaconda"]
C. SyntaxError
D. None of the above

The Answer

C) SyntaxError when ran in Python 3

Explanation

This quiz is tricky because the answer depends on which version of Python you run it in. Technically speaking, the code above is valid in Python 2, so if you run it there, the answer would be **A) 'Caught an error!'**. However, it is buggy: the Python 2 syntax would assign the caught `IndexError` to the name `ValueError`, and any `ValueErrors` raised would not be caught.

Besides, Python 2 development ended in 2020, so you really should only be using it for legacy support.

In Python 3, the answer is **C) SyntaxError**. The syntax in Python 3 changed so that if you want to catch multiple exceptions on a single line, you need to use parentheses.

You can change the code like this:

```python
my_list = ["Python", "Boa", "Anaconda"]
try:
    print(my_list[4])
except (IndexError, ValueError):
    print("Caught an error!")
```

Now that you enclosed the two exceptions in parentheses, the code works, and the answer would now be “A” if you rerun it.

Quiz 80 - Tuple Enumeration

Python's `enumerate()` function is handy. But how does it work when you want to enumerate a list of tuples or a list of lists?

Does `enumerate()` even work at all in that situation?

Here's your time to shine and show what you know about `enumerate()`!

The Quiz

Which of the following is the correct way to enumerate the tuple?

```python
list_of_tuples = [("snake", "Python"), ("rodent", "hamster")]

# A)
for i, genus, animal in enumerate(list_of_tuples):
    print(f"{i} {genus = } {animal =}")

# B)
for i, (genus, animal) in enumerate(list_of_tuples):
    print(f"{i} {genus = } {animal =}")

# C)
for genus, animal in enumerate(list_of_tuples):
    print(f"{i} {genus = } {animal =}")

# D) None of the above
```

Hint

You can always read the Python documentation[55] for `enumerate()`.

[55]https://docs.python.org/3/library/functions.html#enumerate

Answer 80 - Tuple Enumeration

This is the answer to **Quiz 80 - Tuple Enumeration**.

Your original question was:

Which of the following is the correct way to enumerate the tuple?

```python
list_of_tuples = [("snake", "Python"), ("rodent", "hamster")]

# A)
for i, genus, animal in enumerate(list_of_tuples):
    print(f"{i} {genus = } {animal =}")

# B)
for i, (genus, animal) in enumerate(list_of_tuples):
    print(f"{i} {genus = } {animal =}")

# C)
for genus, animal in enumerate(list_of_tuples):
    print(f"{i} {genus = } {animal =}")

# D) None of the above
```

The Answer

```python
# B)
for i, (genus, animal) in enumerate(list_of_tuples):
    print(f"{i} {genus = } {animal =}")
```

Explanation

If you want to get technical about it, both answers "B" and "C" will work, but "C" has a `NameError` in it.

Take a look at the following output from answer "B" first:

```
>>> list_of_tuples = [("snake", "Python"), ("rodent", "hamster")]
>>> for i, (genus, animal) in enumerate(list_of_tuples):
...     print(f"{i} {genus = } {animal =}")
...
0 genus = 'snake' animal ='Python'
1 genus = 'rodent' animal ='hamster'
```

This code accounts for the index as well as both items that are inside the tuple. You need to ensure that you have the same number of variables in the tuple as the nested items you are enumerating.

This code will work if the nested tuples or nested lists are of varying lengths.

Now let's modify answer "C" to remove the reference to the variable, "i", which isn't included in the code:

```
>>> list_of_tuples = [("snake", "Python"), ("rodent", "hamster")]
>>> for genus, animal in enumerate(list_of_tuples):
...     print(f"{genus = } {animal =}")
...
genus = 0 animal =('snake', 'Python')
genus = 1 animal =('rodent', 'hamster')
```

While this code works, it has a logic error. You can see that `genus` is the `index` returned from `enumerate()` while `animal` is the tuple inside the list. This code is a great workaround for having nested lists or tuples of various lengths, although you should rename `genus` to `index` or something similar to make the code clearer.

Quiz 81 - Lists and Strings

Do you know which data types can be combined in Python? Some of them are obvious. For example, floats and integers can be added together.

But what if you wanted to "add" a string and a list? Does that work?

Good luck!

The Quiz

What is the output of this silly code?

```python
my_list = []
my_list += "Python"
print(my_list)
```

A. SyntaxError
B. RuntimeError
C. ['Python']
D. ['P', 'y', 't', 'h', 'o', 'n']

Hint

You should read up on how Python lists do augmented assignment.

Answer 81 - Lists and Strings

This is the answer to **Quiz 81 - Lists and Strings**.

Your original question was:

What is the output of this silly code?

```python
my_list = []
my_list += "Python"
print(my_list)
```

A. SyntaxError
B. RuntimeError
C. [‘Python’]
D. [‘P’, ‘y’, ‘t’, ‘h’, ‘o’, ‘n’]

The Answer

D) [‘P’, ‘y’, ‘t’, ‘h’, ‘o’, ‘n’]

Explanation

Python lists process augmented assignment in an interesting way. The list will iterate over the items on the right side of the augmented assignment operator and add each item individually to the list on the left-hand side.

Here are a couple of examples:

```python
# Augmented assignment using two lists
>>> my_list = [1, 2, 3]
>>> my_list += [4, 5, 6]
>>> my_list
[1, 2, 3, 4, 5, 6]

# Augmented assignment using a list and a tuple
>>> my_list = [1, 2, 3]
>>> my_list += (10, 11, 12)
>>> my_list
[1, 2, 3, 10, 11, 12]
```

Strings in Python are iterable. When you use augmented assignment with a list and a string, the string is added to the list one character at a time.

Quiz 82 - Multireference Anarchy

Mutable objects are so much fun! This quiz will teach you what happens when two variables are set to the same list in Python.

When you change one reference, does it change the other? Or is the second variable a copy of the first?

You might find out in this quiz!

The Quiz

What will be the output of the following code?

```python
numbers = [1, 2, 3]
integers = numbers
numbers = numbers + [4, 5, 6]
print(integers)
```

A. [1, 2, 3]
B. [4, 5, 6]
C. [1, 2, 3, 4, 5, 6]
D. []

Hint

Read the code very carefully!

Answer 82 - Multireference Anarchy

This is the answer to **Quiz 82 - Multireference Anarchy**.

Your original question was:

What will be the output of the following code?

```python
numbers = [1, 2, 3]
integers = numbers
numbers = numbers + [4, 5, 6]
print(integers)
```

A. [1, 2, 3]
B. [4, 5, 6]
C. [1, 2, 3, 4, 5, 6]
D. []

The Answer

A) [1, 2, 3]

Explanation

What happened here? Well, if you look at the third line of code, you can see that you are resetting `numbers` to a new value:

```python
numbers = numbers + [4, 5, 6]
```

The `numbers` variable is now assigned the value of the original `numbers` plus the values in `[4, 5, 6]`.

You may be able to understand this better by taking a look at the following REPL session:

```
>>> numbers = [1, 2, 3]
>>> id(numbers)
140211309157568
>>> integers = numbers
>>> id(integers)
140211309157568
>>> integers is numbers
True

# Reassign numbers to something new
>>> numbers = numbers + [4, 5, 6]

# The id changes
>>> id(numbers)
140211317177856

# now integers and numbers no longer refer to the same object
>>> integers is numbers
False
```

If you change the code slightly to use augmented assignment, the result will be different:

```
>>> numbers = [1, 2, 3]
>>> integers = numbers
>>> numbers += [4, 5, 6]
>>> numbers
[1, 2, 3, 4, 5, 6]
>>> integers
[1, 2, 3, 4, 5, 6]
```

Here you use augmented assignment, `+=`, instead of regular assignment, `=`, and since `lists` are mutable it is changed in-place instead of creating a new `list`. Another way to get the same result is to assign to the list itself, instead of the name:

```
>>> numbers = [1, 2, 3]
>>> integers = numbers
>>> numbers[:] = numbers + [4, 5, 6]
>>> numbers
[1, 2, 3, 4, 5, 6]
>>> integers
[1, 2, 3, 4, 5, 6]
```

Quiz 83 - Tuples and Dictionaries

You might be getting sick of quizzes about Python data types. However, data types are very important in all programming languages. You need to understand how your language has implemented their data types.

Take the dictionary for instance. A dictionary can be created in several different ways. You can use curly braces (i.e. `{}`) or you can use the `dict()` callable.

For this quiz, you will need to know how `dict()` works!

The Quiz

What is the output of this code?

```python
print(dict(("py", "th", "on")))
```

A. SyntaxError
B. TypeError
C. `{"py": None, "th": None, "on": None}`
D. `{'p': 'y', 't': 'h', 'o': 'n'}`

Hint

You should research what the `dict()` class takes as arguments.

Answer 83 - Tuples and Dictionaries

This is the answer to **Quiz 83 - Tuples and Dictionaries**.

Your original question was:

What is the output of this code?

```python
print(dict(("py", "th", "on")))
```

A. SyntaxError
B. TypeError
C. `{"py": None, "th": None, "on": None}`
D. `{'p': 'y', 't': 'h', 'o': 'n'}`

The Answer

D) `{'p': 'y', 't': 'h', 'o': 'n'}`

Explanation

Your first stop in understanding how this works is to look at the documentation for dict()[56]. The `dict()` callable can take in any of the following:

- mapping (another dict, basically)
- iterable
- Nothing at all

In this quiz, you pass in a tuple of three two-letter strings. That's important! The `dict()` wouldn't work if the strings were longer or shorter.

Check it out:

[56]https://docs.python.org/3/library/functions.html#func-dict

```
>>> dict(('abc', 'def', 'ghi'))
Traceback (most recent call last):
  Python Shell, prompt 10, line 1
builtins.ValueError: dictionary update sequence element #0 has length 3; 2 is requir\
ed
>>> dict(('a', 'b', 'c'))
Traceback (most recent call last):
  Python Shell, prompt 11, line 1
builtins.ValueError: dictionary update sequence element #0 has length 1; 2 is requir\
ed
```

In fact, the exception that is raised tells you that the sequence length has to be two!

But why? Well, when you create a dictionary, what do you need? You need a key and a value. That's two items of information!

When you pass in your tuple, it must contain one or more two-item sequences. For this quiz, your sequence was a two-letter string.

The first letter is the key and the second letter is the value. When you run the code, you end up with this dictionary: `{'p': 'y', 't': 'h', 'o': 'n'}`

Quiz 84 - Ridiculous Rounding

What is rounding anyway? One of the most common types of rounding is to round to the next highest number if the decimal point is greater than or equal to five. Otherwise, you round down.

The question in this quiz is to determine what type of rounding Python does with its built-in `round()` function?

Do you know? If you do, then this quiz will be easy for you!

The Quiz

Does the following code print True or False?

```python
# True or False?

print(round(1.5) == round(2.5))
```

Hint

If you get stuck, you'll want to read the documentation[57].

[57] https://docs.python.org/3/library/functions.html#round

Answer 84 - Ridiculous Rounding

This is the answer to **Quiz 84 - Ridiculous Rounding**.

Your original question was:

Does the following code print True or False?

```python
# True or False?

print(round(1.5) == round(2.5))
```

The Answer

The answer is **True**!

Explanation

Python 3 uses “Banker’s Rounding” which is defined like this: “Exact halfway cases are now rounded to the nearest even result instead of away from zero” - per What’s new in Python 3[58].

In the case of 1.5, Python rounds to two because that is the nearest even result. If you apply that logic to 2.5, the nearest even value is **also** two!

So when you check the equality of rounding 1.5 and 2.5, you’ll see that the result is `True`.

[58]https://docs.python.org/3/whatsnew/3.0.html#builtins

Quiz 85 - Conditional Unpacking

You have assigned multiple values to multiple variables before. This is called **Unpacking Generalizations** in Python. You can read more about this concept in PEP 448[59].

Today you will take an unpacking generalization and combine it with the humble ternary operator. A ternary operator is a fancy term for a one-line conditional statement.

Good luck!

The Quiz

What is the output of the following code?

```python
x, y = (0, 1) if True else None, None
print(x)
```

A. 0
B. 1
C. None
D. (0, 1)

Hint

Study the code closely. If you get stuck, you can read all about conditional expressions in PEP 308[60].

[59]https://peps.python.org/pep-0448/
[60]https://peps.python.org/pep-0308/

Answer 85 - Conditional Unpacking

This is the answer to **Quiz 85 - Conditional Unpacking**.

Your original question was:

What is the output of the following code?

```python
x, y = (0, 1) if True else None, None
print(x)
```

A. 0
B. 1
C. None
D. (0, 1)

The Answer

D) (0, 1)

Explanation

This quiz is goofy. You can't easily see the answer without adding some additional parentheses. But before you do that, check what's in `y`:

```python
>>> x, y = (0, 1) if True else None, None
>>> x
(0, 1)
>>> print(y)
None
```

The `y` variable has `None` in it! How did that happen? The last `None` in the ternary is assigned to `y`. But why is that happening?

The reason is that the last `None` isn't a part of the ternary at all!

The ternary is as follows: `(0, 1) if True else None`. That's it!

Try running that in your REPL session:

```
>>> (0, 1) if True else None
(0, 1)
```

That code will always return `(0, 1)` because your conditional is silly. The conditional says "if True", which is always True. It's like when you create an infinite `while` loop using the `while True:` syntax.

Now you can add the extra parentheses from earlier:

```
>>> x, y = ((0, 1) if True else None), None
>>> x
(0, 1)
>>> print(y)
None
```

This code is equivalent to the following:

```
>>> x, y = (0, 1), None
>>> x
(0, 1)
>>> print(y)
None
```

Use this knowledge wisely and don't write silly code like this!

Quiz 87 - List Addition

You probably know this already, but Python lets you do mathematical expressions. You can add or multiply numbers, for example.

If you've been paying attention in this book, you will have noticed that Python lets you add other data types together too. An excellent example is using the plus symbol to combine two or more strings.

But what happens when you add lists together? That will be your challenge today!

The Quiz

What is the output of the following code?

```python
i = j = [3]
i += j
print(i, j)
```

A. [3]
B. [3, 3]
C. [3, 3] [3, 3]
D. [3, 3, 3, 3]

Hint

This is one of those quizzes where using the REPL to experiment is a good idea.

Answer 87 - List Addition

This is the answer to **Quiz 87 - List Addition**.

Your original question was:

What is the output of the following code?

```python
i = j = [3]
i += j
print(i, j)
```

A. [3]
B. [3, 3]
C. [3, 3] [3, 3]
D. [3, 3, 3, 3]

The Answer

C) [3, 3] [3, 3]

Explanation

When you add two lists together, you extend the list.

Here is an example:

```python
>>> list_a = [1, 2, 3]
>>> list_b = [4, 5, 6]
>>> list_a + list_b
[1, 2, 3, 4, 5, 6]
```

In your quiz, you set two variables to the same list, and then you add the list to itself:

```
>>> i = j = [3]
>>> i
[3]
>>> j
[3]
>>> i += j
>>> i
[3, 3]
```

Finally, you want to print out `i` and `j`, which are the same object. In other words, you print the same thing twice:

```
>>> print(i, j)
[3, 3] [3, 3]
```

When you want to print multiple objects in Python, you separate each object with a comma. However, the commas themselves are **not** printed in the output.

Quiz 88 - Essential Assertions

Python comes with testing capabilities built-in. You can use the `unittest` module to write unit tests for your code. A popular alternative to the native testing module is pytest[61].

No matter which testing framework you choose, you will use some assert. The `unittest` module has its special asserts, while pytest uses Python's built-in `assert` keyword.

To solve this quiz, you need to know the proper way to write an assert. Good luck!

The Quiz

What is the output of this assert?

```
a = "python"
b = "javascript"
assert(a==b, "Languages are different")
```

A. True
B. False
C. SyntaxWarning
D. SyntaxError

Hint

You may find reviewing the assert documentation[62] helpful.

[61]https://docs.pytest.org/en/7.2.x/
[62]https://docs.python.org/3/reference/simple_stmts.html#the-assert-statement

Answer 88 - Essential Assertions

This is the answer to **Quiz 88 - Essential Assertions**.

Your original question was:

What is the output of this assert?

```
a = "python"
b = "javascript"
assert(a==b, "Languages are different")
```

A. True
B. False
C. SyntaxWarning
D. SyntaxError

The Answer

C) SyntaxWarning

Explanation

While the assert as written is syntactically correct, the assert itself is lousy.

The reason the assert is lousy is that the parentheses make the assert always return True.

If you run this code, you will see the following output:

```
SyntaxWarning: assertion is always true, perhaps remove parentheses?
  assert(a==b, "Languages are different")
```

That output gives you a clue that you've done something incorrectly.

Try removing the parentheses and rerunning it:

```
>>> a = "python"
>>> b = "javascript"
>>> assert a == b, "Languages are different"
Traceback (most recent call last):
  Python Shell, prompt 80, line 1
builtins.AssertionError: Languages are different
```

You get a proper `AssertionError` with your custom message when you run this code.

If you run it without the custom message, the output will look like this:

```
>>> assert a == b
Traceback (most recent call last):
  Python Shell, prompt 81, line 1
builtins.AssertionError
```

Pro tip: Always be aware of the proper syntax when you write code.

Quiz 89 - String Silliness

You learned about list addition in a recent quiz. Now you'll learn about string addition, also known as string concatenation.

Concatenation is a fancy word for connecting two strings into one new string. You can use Python's `id()` function to verify that each of the three string objects has a different identity.

But this test isn't about identity or equality, so don't let those thoughts muddy your thinking.

Instead, try to figure out this quiz!

The Quiz

What is the output of the following code?

```python
x = y = "Python"
x += "rocks!"
print(x, y)
```

A. Python
B. Python rocks!
C. Python rocks! Python
D. Pythonrocks! Python

Hint

What happens when you concatenate two strings together anyway?

Answer 89 - String Silliness

This is the answer to **Quiz 89 - String Silliness**.

Your original question was:

What is the output of the following code?

```python
x = y = "Python"
x += "rocks!"
print(x, y)
```

A. Python
B. Python rocks!
C. Python rocks! Python
D. Pythonrocks! Python

The Answer

D) Pythonrocks! Python

Explanation

What is happening here? When you do augmented assignment of "rocks!" to the `x` variable, it changes what `x` is assigned to, but it does **not** update `y`.

Take a look:

```python
>>> x = y = "Python"
>>> x is y
True
>>> x += "rocks!"
>>> x
'Pythonrocks!'
>>> y
'Python'
```

You can see from this REPL session that at the beginning of the code, not only are `x` and `y` equal in value, but their identity is the same too. However, after you use augmented assignment (+=), `x` has changed and `y` has not, because `str`'s are immutable and cannot be changed.

The last piece of code from the quiz is to print out `x` and `y`. You don't add a space between the two variables, so `print()` concatenates the strings accordingly and sends the result to the terminal.

Quiz 90 - List Insertion

Python’s list object has many methods you can use. It would be best if you took the time to learn the common methods, as they can prove invaluable in your coding journey.

You will be quizzed on your understanding of the `insert()` method. Do you know?

Do your best!

The Quiz

What is the output of the following code?

```python
my_list = list("python")
my_list.insert(100, "!")
print(my_list)
```

A. [‘p’, ‘y’, ‘t’, ‘h’, ‘o’, ‘n’, ‘!’]
B. SyntaxError
C. IndexError
D. [’!’, ‘p’, ‘y’, ‘t’, ‘h’, ‘o’, ‘n’]

Hint

Should you get stuck, you can look at the list documentation[63] for some ideas.

[63]https://docs.python.org/3/tutorial/datastructures.html

Answer 90 - List Insertion

This is the answer to **Quiz 90 - List Insertion**.

Your original question was:

What is the output of the following code?

```python
my_list = list("python")
my_list.insert(100, "!")
print(my_list)
```

A. [‘p’, ‘y’, ‘t’, ‘h’, ‘o’, ‘n’, ‘!’]
B. SyntaxError
C. IndexError
D. [’!’, ‘p’, ‘y’, ‘t’, ‘h’, ‘o’, ‘n’]

The Answer

A) [‘p’, ‘y’, ‘t’, ‘h’, ‘o’, ‘n’, ‘!’]

Explanation

The `insert()` method of the list data structure takes two arguments:

- index - The location to insert an element
- The item to insert into the list

You want to insert an exclamation point at the one-hundredth position in this example. The list only has six items in it.

What happens when you insert a character at a position that does not exist in a list? If the position is greater than the length of the list, the item is inserted at the end of the list.

If you used a negative value less than the length of the string, the value would be inserted starting from the end of the list. If the negative value exceeds negative six, the character will be inserted at the beginning of the list.

Here is an example using a negative value less than negative six:

```
>>> my_list = list("python")
>>> my_list.insert(-1, "!")
>>> my_list
['p', 'y', 't', 'h', 'o', '!', 'n']
```

And here is an example of using a negative value greater than negative six:

```
>>> my_list = list("python")
>>> my_list.insert(-100, "!")
>>> my_list
['!', 'p', 'y', 't', 'h', 'o', 'n']
```

At this point, you should understand how the list's `insert()` method works.

Quiz 91 - Letter Splitting

Python strings have many methods too. Most of Python's built-in objects or data types have many methods. Strings may have the most methods of any of the built-in data types!

Fortunately, you do not need to know every string method to use Python effectively. However, it is good to know a few.

In this quiz, you will look at the `split()` method. What does that do when a string has only one character?

Good luck!

The Quiz

What is the output of this code?

```python
print("a".split())
```

A. a
B. []
C. ['a']
D. None of the above

Hint

If you need help, you should read the documentation on this string method[64].

[64]https://docs.python.org/3/library/stdtypes.html#str.split

Answer 91 - Letter Splitting

This is the answer to **Quiz 91 - Letter Splitting**.

Your original question was:

What is the output of this code?

```python
print("a".split())
```

A. a
B. []
C. [‘a’]
D. None of the above

The Answer

C) [‘a’]

Explanation

The string’s `split()` method takes an optional `sep` parameter. The `sep` parameter is the delimiter you want to split on, and it defaults to whitespace. You can also set the maximum number of splits using the `maxsplit` parameter.

Here are a few examples of using the `split()` method:

```python
>>> 'abcd'.split()
['abcd']
>>> 'abcd'.split('b')
['a', 'cd']
>>> 'a,b,c,d'.split(',')
['a', 'b', 'c', 'd']
>>> 'a,b\t \nc,d'.split()
['a,b', 'c,d']
```

As you can see, if you do not provide a delimiter to split on, groupings of whitespace are used. If the string contains no whitespace, the entire string is added to the list; otherwise, you split the string on the delimiter provided and get a list with one or more substrings.

You do not provide a delimiter for this quiz, so the entire string is added to the list and returned.

Quiz 92 - Class Attributes

Python supports multiple types of attributes. You have class attributes and instance attributes. A class attribute is an attribute you access using the class itself, and an instance of a class can access both class and instance attributes.

Your challenge in this quiz is to understand scope and class attributes.

Have fun!

The Quiz

What is the output of the following code?

```python
letters = ["a"]

class L:
    letters = ["b"]
    letters = letters + ["c"]

print(letters[0], L.letters)
```

A. a [‘b’, ‘c’]
B. [‘a’, ‘b’, ‘c’]
C. [‘b’, ‘c’]
D. None of the above

Hint

You can review class in The Python tutorial[65].

[65] https://docs.python.org/3/tutorial/classes.html

Answer 92 - Class Attributes

This is the answer to **Quiz 92 - Class Attributes**.

Your original question was:

What is the output of the following code?

```python
letters = ["a"]

class L:
    letters = ["b"]
    letters = letters + ["c"]

print(letters[0], L.letters)
```

A. a [‘b’, ‘c’]
B. [‘a’, ‘b’, ‘c’]
C. [‘b’, ‘c’]
D. None of the above

The Answer

A) a [‘b’, ‘c’]

Explanation

This quiz is pretty silly. You shadow the global `letters` variable inside the class by creating a class attribute of the same name.

Because of this, you print out the first letter in your `letters` list (not in the class), and then the contents of the `letters` class attribute.

For fun, you can rewrite the code a little and see if you can access the `letters` variable inside the class:

```
>>> letters_out = ["a"]
>>> class L:
...     letters = ["b"]
...     letters = letters + ["c"]
...     letters_out.append(letters)
...
>>> letters_out
['a', ['b', 'c']]
```

Here you rename the outside variable to `letters_out` and then attempt to append to it inside of the class. Finally, you try to print out the contents of `letters_out`.

As you can see, Python immediately evaluated the class's contents after you created it, and you did not need to instantiate the class for that to happen. Because Python does this, Python appended the inner `letter` list to your `letters_out` list.

This type of programming is typically discouraged because it needs to be clarified. It would be best if you avoided it, but it is good to know how Python behaves, so you don't get tripped up by code like this.

Quiz 93 - Class For Loop

You will continue your adventures with class attributes in this quiz.

You will need to ask what happens to the variable in a `for` loop when the `for` loop is at the class attribute level? Is that variable available as a class attribute or not?

Once you've answered that question, you should be able to determine the answer!

The Quiz

What is the output of the following code?

```python
class Number:
    integers = [5, 6, 7]
    for i in integers:
        i * 2

print(Number.i)
```

A. AttributeError
B. 10
C. 14
D. 7

Hint

As usual, study the code closely, and the answer will reveal itself!

Answer 93 - Class For Loop

This is the answer to **Quiz 93 - Class For Loop**.

Your original question was:

What is the output of the following code?

```python
class Number:
    integers = [5, 6, 7]
    for i in integers:
        i * 2

print(Number.i)
```

A. AttributeError
B. 10
C. 14
D. 7

The Answer

D) 7

Explanation

The variable used in the `for` loop exists as a class attribute you can access. The trick is to notice that you are throwing away the multiplied value instead of reassigning it to `i`.

You can modify the code, so you're not multiplying `i` and get the same result:

```
>>> class Number:
...     integers = [5, 6, 7]
...     for i in integers:
...         print(i)
...
5
6
7
>>> print(Number.i)
7
```

The variable, `i`, will be whatever the last number is in the list.

Quiz 94 - The Length of a Split

You may have already gotten your fill of string-splitting silliness, but you're not done yet! In this quiz, you'll look at what happens when you split an empty string!

Will `split()` return `None`? Do you get some kind of exception thrown? Or does `split()` return an empty list or what?

Let's see if you know the answer!

The Quiz

What is the length that is returned?

```python
print(len("".split(" ")))
```

A. 0
B. 1
C. 2
D. None of the above

Hint

You can try reading the documentation on this string method[66] or experiment with the code yourself in the REPL.

[66]https://docs.python.org/3/library/stdtypes.html#str.split

Answer 94 - The Length of a Split

This is the answer to **Quiz 94 - The Length of a Split**.

Your original question was:

What is the length that is returned?

```
print(len("".split(" ")))
```

A. 0
B. 1
C. 2
D. None of the above

The Answer

B) 1

Explanation

The code in this quiz is attempting to split on a space. An empty string doesn't have a space, but you get a list with an empty string anyway.

Here's the code in a REPL:

```
>>> "".split(" ")
['']
```

The `split()` method tries to split on a space as the separator.

You will see the same effect if you split using other characters that aren't in the string:

```
>>> "".split("1")
['']
>>> "".split(",")
['']
```

In the real world, check whether the string is empty before you split. But these kinds of bugs can trip you up and cause you to pull your hair out. Always test your code to know what it will do, especially edge cases.

Quiz 95 - Senseless Sorting

As you know, Python has dozens of built-in functions and callables. Today your knowledge of the `sorted()` function will be challenged.

You need to look at this quiz and determine whether the comparison is equal. Isn't that fun?

You will often do this task as an engineer, and you need to know when two objects have equality very often. Another task that goes hand-in-hand with this one is to check if an iterable contains an object.

But that is outside the scope of this quiz. Good luck!

The Quiz

Does the following print `True` or `False`?

```python
x = 7, 8, 9
print(sorted(x) == x)
```

Hint

You will find a clue in the `sorted()` function documentation[67].

[67]https://docs.python.org/3/library/functions.html#sorted

Answer 95 - Senseless Sorting

This is the answer to **Quiz 95 - Senseless Sorting**.

Your original question was:

Does the following print `True` or `False`?

```python
x = 7, 8, 9
print(sorted(x) == x)
```

The Answer

False

Explanation

This code returns `False` because you are comparing two different data types against each other.

- `sorted(x)` returns a list
- `x` is a tuple

Even though the items inside the tuple and the list are in the same order, it returns' False' because you are comparing a list to a tuple.

You can see the difference in the following REPL session:

```python
>>> x = 7, 8, 9
>>> x == (7, 8, 9)
True
>>> y = sorted(x)
>>> type(y)
<class 'list'>
>>> y == x
False
>>> y == [7, 8, 9]
True
```

If you compare sorted tuples to tuples or lists to lists, the result will be `True`. But if you compare sorted lists and tuples, the result will be `False`.

Quiz 96 - Eclectic Ellipses

Do you know everything there is to know about functions in Python? This quiz will help prove your knowledge! Functions are something that you will be writing many times as a software developer. Learning how to break up your code into small, reusable chunks is a valuable tool to have!

To solve this riddle, you must know all the required syntax necessary to create a function. Knowing all the Python types would help too.

Have fun!

The Quiz

What is the output of the following code?

```python
def my_function():
    Ellipsis

print(my_function())
```

A. SyntaxError
B. None
C. Ellipsis
D. None of the above

Hint

If you get stuck, you may find that the documentation[68] has a clue or two.

[68]https://docs.python.org/3/library/stdtypes.html

Answer 96 - Eclectic Ellipses

This is the answer to **Quiz 96 - Eclectic Ellipses**.

Your original question was:

What is the output of the following code?

```python
def my_function():
    Ellipsis

print(my_function())
```

A. SyntaxError
B. None
C. Ellipsis
D. None of the above

The Answer

B) None

Explanation

`Ellipsis` is a built-in name. If you call `Ellipsis`, you will get the `Ellipsis` singleton returned.

But that's neither here nor there. The key to understanding this code is that you created a valid function by putting something in the function itself, and you could have just as easily put in the `print` keyword or called it, and the code still would have worked.

If you don't include any function contents, you'll end up with an `IndentationError` or another type of exception.

In this example, you don't specify a return value. All functions in Python return `None` by default unless you return something else explicitly.

So, the answer to this quiz is: **B) None**

Quiz 98 - Diabolical Decorator

A decorator, at its simplest, is a function that accepts another function as its argument and does something interesting, such as:

- adding one or more new features without affecting how the function works
- registering the function with another data structure
- adding pre- or post-processing when you call the function

Everyday use cases for decorators include:

- logging
- retrying
- security
- formatting
- debugging

Decorators have been a part of Python for a long time, but they tend to trip up beginners. This quiz will test your knowledge of decorators and how they work.

Enjoy!

The Quiz

What does this code print out in Python 3.9+?

```python
class dog:
    ...

@(lambda f: setattr(dog, "on_leash", f))
def walk():
    print("bark bark!")

print(dog.on_leash())
```

A. SyntaxError
B. bark bark!

None

C. bark bark!
D. None of the above

Hint

If you get stuck, you might want to read PEP 614[69].

[69]https://peps.python.org/pep-0614/

Answer 98 - Diabolical Decorator

This is the answer to **Quiz 98 - Diabolical Decorator**.

Your original question was:

What does this code print out in Python 3.9+?

```python
class dog:
    ...

@(lambda f:setattr(dog, "on_leash", f))
def walk():
    print("bark bark!")

print(dog.on_leash())
```

A. SyntaxError
B. bark bark!

None

C. bark bark!
D. None of the above

The Answer

B) bark bark!
None

Explanation

Starting in Python 3.9, you can now use Python lambdas as decorators. You can use other built-in Python functions as decorators too.

The example in this quiz may make your Python IDE mark the lambda line as a syntax error even though the code runs.

This quiz demonstrates a type of monkey-patching. A monkey patch is a way to update a piece of code at runtime dynamically. You can see that the `dog` class does not have an `on_leash()` method. However, you add the `on_leash()` method using the decorator at runtime.

A slightly better way of writing this would have been something like this:

```python
class dog:
    ...


def enhance_dog(func):
    setattr(dog, "on_leash", func)

@enhance_dog
def walk():
    print("bark bark!")

print(dog.on_leash())
```

A more generic version could look like this:

```python
class dog:
    ...

class add_cls_method:
    def __init__(self, cls):
        self.cls = cls
    def __call__(self, func):
        setattr(self.cls, func.__name__, func)

@add_cls_method(dog)
def walk():
    print("bark bark!")

print(dog.walk())
```

All three of these decorators have one serious flaw: the name `walk` is assigned to whatever the decorator returns, and all three return `None`. If you try to call `walk()` directly, you'll get a `TypeError` because `None` is not callable. The last step of any decorator should be to return either the original function, or a replacement function.

Quiz 99 - Tuple Comparison

You may have thought you were done with these crazy comparison quizzes, but you have a new challenge today. How do you compare tuples? Do they compare the way you expect?

Take a look at the quiz and spend some time studying the code. If you do, you're sure to solve it!

Have fun!

The Quiz

What is the output of the following code?

```python
x = 7, 8, 9
print(x == 7, 8, 9)
```

A. True
B. False
C. (7, 8, 9)
D. None of the above

Hint

It would help if you understood precedence and function arguments to solve this one.

Answer 99 - Tuple Comparison

This is the answer to **Quiz 99 - Tuple Comparison**.

Your original question was:

What is the output of the following code?

```python
x = 7, 8, 9
print(x == 7, 8, 9)
```

A. True
B. False
C. (7, 8, 9)
D. None of the above

The Answer

D) None of the above

Explanation

The output of this code is: **False 8 9**

What's going on here? When you call a function, the arguments are separated by commas, so you are actually sending three arguments to `print()`:

- `x == 7`
- `8`
- `9`

if you only want to send one argument, you need more parentheses:

`print(x == (7, 8, 9))`

Keep in mind: commas (`,`) create tuples; parentheses (`()`) are for grouping – except for an empty tuple, which is created like this: `()`.

Quiz 100 - Idiomatic Inheritance

Python classes support the concept of inheritance. Python supports multiple inheritance, which means that one class can inherit methods and attributes from more then one base class.

It would be best to be careful with inheritance as it can make your code harder to follow. There are examples of multiple inheritance in Django, PyQt and wxPython that you can look at if you want to see real-world examples.

For this quiz, you will focus on single inheritance. The code is more straight-forward and easy enough for you to glance at the code and figure out the answer quickly.

Good luck!

The Quiz

What is the output of the following code?

```python
class Py:
    py = 1

class Cy(Py):
    ...

class Vy(Py):
    ...

Cy.py = 3.14
Py.py = 7
print(f"{Py.py = } {Cy.py = } {Vy.py = }")
```

A. Py.py = 7 Cy.py = 3.14 Vy.py = 7
B. Py.py = 7 Cy.py = 3.14 Vy.py = 3.14
C. Py.py = 7 Cy.py = 3.14 Vy.py = 1
D. None of the above

Hint

A good place to learn more about classes and inheritance is in the Python documentation[70].

[70]https://docs.python.org/3/tutorial/classes.html

Answer 100 - Idiomatic Inheritance

This is the answer to **Quiz 100 - Idiomatic Inheritance**.

Your original question was:

What is the output of the following code?

```python
class Py:
    py = 1

class Cy(Py):
    ...

class Vy(Py):
    ...

Cy.py = 3.14
Py.py = 7
print(f"{Py.py = } {Cy.py = } {Vy.py = }")
```

A. Py.py = 7 Cy.py = 3.14 Vy.py = 7
B. Py.py = 7 Cy.py = 3.14 Vy.py = 3.14
C. Py.py = 7 Cy.py = 3.14 Vy.py = 1
D. None of the above

The Answer

A) Py.py = 7 Cy.py = 3.14 Vy.py = 7

Explanation

The base class in this code is the `Py` class. The `Py` class has a single class attribute, `py`. Note that the attribute is lowercase while the class has a capital "P".

The classes `Cy` and `Vy`, are subclasses or child classes of the `Py` class. You can tell that because each of these classes has `Py` as their first argument. Both `Cy` and `Vy` are empty, so they don't add any attributes or methods or override the `py` attribute itself. That means they inherit whatever `py` is set to in the base class.

The last three lines of code change some things:

```python
Cy.py = 3.14
Py.py = 7
print(f"{Py.py = } {Cy.py = } {Vy.py = }")
```

Here you see that you set`Cy.py` to 3.14, while `Py.py` is set to 7. `Py` is the base class, so that affects `Vy`, which is not being set explicitly. However, it doesn't override `Cy` because `Cy.py` was set to something new.

If you hadn't set `Cy.py`, then setting `Py.py` would have changed all three classes.

So what you end up with in the end is that `Py.py` and `Vy.py` are set to seven while `Cy.py` is set to 3.14.

Quiz 101 - Maniacal Munging

Python code is usually thought of as easy to read and write. Some developers from other languages to Python ask about how Python does private and protected class members or attributes.

Python doesn't have the concept of protected attributes outside of perhaps using an Abstract Base Class. But what about private attributes? Python *does* kind of support those by using leading underscores.

Your challenge is to figure out how these "private" attributes work!

The Quiz

What is the output of the last line in this REPL session?

```
>>> class Dog:
...     def __init__(self):
...         self.__color = "blue"
...         self.hair = "long"
...
>>> Dog().hair
'long'
>>> Dog().__color
???
```

A. 'blue'
B. SyntaxError
C. AttributeError
D. None of the above

Hint

If you get stuck, you can do an Internet search for the term "name mangling".

Answer 101 - Maniacal Munging

This is the answer to **Quiz 101 - Maniacal Munging**.

Your original question was:

What is the output of the last line in this REPL session?

```
>>> class Dog:
...     def __init__(self):
...         self.__color = "blue"
...         self.hair = "long"
...
>>> Dog().hair
'long'
>>> Dog().__color
???
```

A. 'blue'
B. SyntaxError
C. AttributeError
D. None of the above

The Answer

C) AttributeError

Explanation

The leading double-underscore in the `__color` attribute causes Python's name mangling mechanism to run. When that happens, Python prefixes the attribute's actual name with `_classname`, so `__color` becomes `_Dog__color`.

Let's have a look at some code in a REPL session, and all will be made clear:

```
>>> class Dog:
...     def __init__(self):
...         self.__color = "blue"
...         self.hair = "long"
...
>>> Dog().__color
Traceback (most recent call last):
  Python Shell, prompt 51, line 1
builtins.AttributeError: 'Dog' object has no attribute '__color'
>>> Dog()._Dog__color
'blue'
```

There is an important note that needs to be said here. The name mangling will only occur if you use a single underscore.

Let's modify this example and check it out:

```
>>> class Dog:
...     def __init__(self):
...         self.__color = "blue"
...         self._hair = "long"
...
>>> Dog()._hair
'long'
>>> dir(Dog())
['_Dog__color', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq_\
_', '__format__', '__ge__', '__getattribute__',
 '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__modul\
e__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', \
'__subclasshook__', '__weakref__', '_hair']
```

If you run Python's `dir()` command on a `Dog` instance, you can see that `__color` was mangled, and `_hair` was not.

The lesson here is that Python uses underscores to mark an attribute as private, but it's only a convention – the attributes/methods are still accessible to other code. The usual premise is that there are no guarantees that underscored names will still be present or work the same in future software versions.

Quiz 102 - Plonking Pinnipeds

Walruses are wonderful animals, one of the largest types of seals, and they look both silly and intimidating.

The Python programming language gets its name from the Monty Python comedy troupe. That may be why Python added a funny little syntax called the “walrus operator” to the language.

Or maybe not. But whatever the reason, you’ll need to know what that is to solve this riddle!

The Quiz

What is the output of the following code?

```python
print((a := 5, 10) == ((a := 5), 10))
```

A. True
B. False
C. SyntaxError
D. None of the above

Hint

If you need to read up on assignment expressions, you will find PEP 572[71] has all the information you’ll need.

[71] https://peps.python.org/pep-0572/

Answer 102 - Plonking Pinnipeds

This is the answer to **Quiz 102 - Plonking Pinnipeds**.

Your original question was:

What is the output of the following code?

```
print((a := 5, 10) == ((a := 5), 10))
```

A. True
B. False
C. SyntaxError
D. None of the above

The Answer

A) True

Explanation

The easiest way to understand this weird code is to run each part of the pieces in the comparison separately.

In other words, run the left side and then the right side in a REPL session:

```
>>> (a := 5, 10)
(5, 10)
>>> ((a := 5), 10)
(5, 10)
>>> (a := 5, 10) == ((a := 5), 10)
True
```

When you run the code this way, you can see that both the left and right sides evaluate to (5, 10), so when you compare them to each other, the result is `True`.

Quiz 103 - Dataclass Difficulties

Python added Dataclasses in version 3.7. They are regular classes with a decorator, `@dataclass`, that adds the `__init__()`, `__repr__()`, and `__eq__()` methods automatically.

They are similar to the popular third-party attrs[72] package. If you have yet to try that package, you should check it out.

The point of this quiz is to help you learn how to write a dataclass by fixing the error in the quiz.

The Quiz

Here you have a `dataclass` where you want a default list. But it doesn't work! Can you fix the code, so it works?

```
>>> from dataclasses import dataclass, field
>>> @dataclass
... class Ford:
...     models: list = field(default_factory=["F-150", "Mustang"])
...
>>> vehicles = Ford()
Traceback (most recent call last):
  Python Shell, prompt 67, line 1
    # Used internally for debug sandbox under external interpreter
  File "<string>", line 3, in __init__
builtins.TypeError: 'list' object is not callable
```

Good luck!

Hint

The dataclass documentation[73] is your first stop in solving this problem if you get stuck.

[72] https://www.attrs.org/en/stable/
[73] https://docs.python.org/3/library/dataclasses.html

Answer 103 - Dataclass Difficulties

This is the answer to **Quiz 103 - Dataclass Difficulties**.

Your original question was:

Here you have a `dataclass` where you want a default list. But it doesn't work! Can you fix the code, so it works?

```
>>> from dataclasses import dataclass, field
>>> @dataclass
... class Ford:
...     models: list = field(default_factory=["F-150", "Mustang"])
...
>>> vehicles = Ford()
Traceback (most recent call last):
  Python Shell, prompt 67, line 1
    # Used internally for debug sandbox under external interpreter
  File "<string>", line 3, in __init__
builtins.TypeError: 'list' object is not callable
```

Good luck!

The Answer

See the next section!

Explanation

The `default_factory` wants some callable to correctly make the `dataclass` function.

One way to resolve this issue is to use an anonymous function, also known as a `lambda`:

```
>>> from dataclasses import dataclass, field
>>> @dataclass
... class Ford:
...     models: list = field(default_factory=lambda: ["F-150", "Mustang"])
...
>>> vehicles = Ford()
>>> vehicles
Ford(models=['F-150', 'Mustang'])
```

If you want the models in the dataclass to be immutable, you can use the `default` parameter with a tuple rather than using `default_factory`:

```
>>> from dataclasses import dataclass, field
>>> @dataclass
... class Ford:
...     models: list = field(default = ("F-150", "Mustang"))
...
>>> vehicles = Ford()
>>> vehicles
Ford(models=('F-150', 'Mustang'))
```

Another potential solution would be to remove the default vehicle models and set the factory to a `list` datatype:

```
>>> @dataclass
... class Ford:
...     models: list = field(default_factory=list)
...
>>> vehicles = Ford()
>>> vehicles
Ford(models=[])
>>> vehicles.models = ["F-150", "Mustang"]
>>> vehicles
Ford(models=['F-150', 'Mustang'])
```

Now you can add whatever vehicle models you want to the dataclass.

Quiz 104 - Cascading Comparisons

Python allows the developer to make multiple comparisons at once without using Boolean operators like `or` and `and`.

You can use this feature to simplify your code if you code carefully.

But these cascading comparisons can sometimes be difficult to parse and understand.

Your challenge is to figure out what this comparison code is doing and what it returns!

The Quiz

What is the output of this code?

```python
print(False == False in [False])
```

A. SyntaxError
B. True
C. False
D. None

Hint

If you get stuck, the Python documentation on comparisons[74] may help.

[74]https://docs.python.org/3/reference/expressions.html#comparisons

Answer 104 - Cascading Comparisons

This is the answer to **Quiz 104 - Cascading Comparisons**.

Your original question was:

What is the output of this code?

```python
print(False == False in [False])
```

A. SyntaxError
B. True
C. False
D. None

The Answer

B) True

Explanation

The key here is understanding how Python does chained or cascading comparisons.

What Python is doing is breaking the code above into the following two comparisons:

```python
>>> False == False
True
>>> False in [False]
True
```

False certainly equals False and False is in the list. Then you put comparison results together, and since they are both True, the result is True.

A more common version of chained comparison is the following: `0 <= x <= 100`. That is easier to read than `x >= 0 and x <= 100`.

Quiz 105 - Methods Into Attributes

Object-oriented programming (OOP) is an important topic to understand. However, Python works well as a functional language too.

You may find this quiz interesting if you are familiar with object-oriented programming.

Your challenge is knowing whether there is a built-in way to convert a method into an attribute. If there is, which one is it?

Good luck!

The Quiz

What do you use to turn the `amount()` method into an attribute?

```python
class Total:

    def __init__(self):
        # private attribute
        self._amount = 10

    def amount(self):
        return self._amount
```

A. @staticmethod
B. @classmethod
C. @property
D. None of the above

Hint

If you need to brush up on those decorators, you must visit Python's Built-in Functions page[75].

[75] https://docs.python.org/3/library/functions.html

Answer 105 - Methods Into Attributes

This is the answer to **Quiz 105 - Methods Into Attributes**.

Your original question was:

What do you use to turn the `amount()` method into an attribute?

```python
class Total:

    def __init__(self):
        # private attribute
        self._amount = 10

    def amount(self):
        return self._amount
```

A. @staticmethod
B. @classmethod
C. @property
D. None of the above

The Answer

C) @property

Explanation

The first step in understanding what this code does is to try accessing `amount` as an instance attribute:

```
>>> class Total:
...
...     def __init__(self):
...         # private attribute
...         self._amount = 10
...
...     def amount(self):
...         return self._amount
...
>>> total = Total()
>>> total.amount  # <-- access amount like an attribute
<bound method Total.amount of <__main__.Total object at 0x7fdc0dd5ebe0>>
>>> total.amount()  # <-- access amount as a method call
10
```

When you access `amount` as an attribute, you get a bound method returned instead of the integer 10. To make the `amount()` behave like an attribute, you can apply the `@property` decorator to it like this:

```
>>> class Total:
...
...     def __init__(self):
...         # private attribute
...         self._amount = 10
...
...     @property
...     def amount(self):
...         return self._amount
...
>>> total = Total()
>>> total.amount
10
```

If you'd like to learn more about properties in Python, check out the official documentation[76].

[76]https://docs.python.org/3/library/functions.html#property

Quiz 106 - Hello World

The "Hello World" program is a common challenge for beginners in a language. The idea is to teach a beginner how to print out the string "Hello World" to their console (AKA standard out).

Today's quiz will be a little more challenging because you have to use two separate `print()` statements to print a single string.

The idea here is to learn more about how `print()` works. Have fun!

The Quiz

How do you make the following code output "Hello Mike" on a single line while keeping **both** `print()` functions?

```python
def hello(name: str) -> None:
    print("Hello")
    print(name)

hello("Mike")
# Expected output: 'Hello Mike' (on a single line)
```

If you run this code, it will print the text incorrectly, like this:

```
Hello
Mike
```

Hint

You'll find some help in the documentation for the built-in print() function[77].

[77]https://docs.python.org/3/library/functions.html#print

Answer 106 - Hello World

This is the answer to **Quiz 106 - Hello World**.

Your original question was:

How do you make the following code output “Hello Mike” on a single line while keeping **both** `print()` functions?

```python
def hello(name: str) -> None:
    print("Hello")
    print(name)

# Expected output: 'Hello Mike' (on a single line)
hello("Mike")
```

The Answer

See the next section!

Explanation

Python’s `print()` function has several optional parameters. Here they are, along with their defaults:

- sep=’ ‘
- end=’\n’
- file=None
- flush=False

You can modify the first `print()` function in your `hello()` function to set the `end` parameter to a string with a single space rather than a carriage return (the default).

Here’s an example:

```
>>> def hello(name: str) -> None:
...     print("Hello", end=" ")
...     print(name)
...
>>> hello("Mike")
Hello Mike
```

Give it a try and see for yourself!

Quiz 107 - String Methodology

Python strings have many methods. How many exactly? Let's take a look:

```
dir("")
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize',
'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal',
'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Not counting the dunder methods, there are forty-seven string methods in total!

You don't need to know all of them, but it would help if you knew a lot of them to solve this riddle!

The Quiz

Which string method do you use to get the expected output?

```
>>> my_str = "FaLse"
>>> print(my_str.???)
fAlSE
```

A. `upper()`
B. `casefold()`
C. `swapcase()`
D. `caseswitch()`

Hint

If you don't know, you can review the string method documentation[78].

[78] https://docs.python.org/3/library/stdtypes.html#string-methods

Answer 107 - String Methodology

This is the answer to **Quiz 107 - String Methodology**.

Your original question was:

Which string method do you use to get the expected output?

```
>>> my_str = "FaLse"
>>> print(my_str.???)
fAlSE
```

A. `upper()`
B. `casefold()`
C. `swapcase()`
D. `caseswitch()`

The Answer

C) `swapcase()`

Explanation

The best way to find out the answer (if you don't know it) is to try each of the given methods out in the REPL:

```
>>> my_str = "FaLse"
>>> my_str.caseswitch()
Traceback (most recent call last):
  Python Shell, prompt 107, line 1
builtins.AttributeError: 'str' object has no attribute 'caseswitch'
>>> my_str.upper()
'FALSE'
>>> my_str.casefold()
'false'
>>> my_str.swapcase()
'fAlSE'
```

The `caseswitch()` method is not a real method. The `upper()` method probably didn't make sense anyway, so you could have ruled that one right out.

The `casefold()` method sounds promising but doesn't do what you want. However, `casefold()` is invaluable for comparing two strings because it works with Unicode strings too.

After exhausting all the other options, the answer is `swapcase()`.

Quiz 108 - Matching Mayhem

If you missed it, the `match` / `case` statement was added in **Python 3.10** and was dubbed Structural Pattern Matching.

Structural Pattern Matching is much more than an alternative for `if / elif / else` in Python.

But rather than bore you with the details, your mission is to figure out what happens when you don't set a specific case in `match / case`. Does Python throw an error or is there more to it than that?

Good luck!

The Quiz

What is the value of `NOT_FOUND` at the end of this code?

```python
return_code = 200
match return_code:
    case NOT_FOUND:
        print("return code not found")

print(f"{NOT_FOUND = }")
```

A. NameError
B. None
C. 200
D. 0

Hint

You can learn more about Structural Pattern Matching in the following PEPs:

- PEP 634 – Structural Pattern Matching: Specification[79]
- PEP 635 – Structural Pattern Matching: Motivation and Rationale[80]
- PEP 636 – Structural Pattern Matching: Tutorial[81]

[79]https://peps.python.org/pep-0634/
[80]https://peps.python.org/pep-0635/
[81]https://peps.python.org/pep-0635/

Answer 108 - Matching Mayhem

This is the answer to **Quiz 108 - Matching Mayhem**.

Your original question was:

What is the value of `NOT_FOUND` at the end of this code?

```python
return_code = 200
match return_code:
    case NOT_FOUND:
        print("return code not found")

print(f"{NOT_FOUND = }")
```

A. NameError
B. None
C. 200
D. 0

The Answer

C) 200

Explanation

When Python runs the `match` code, it checks if the `return_code` matches by binding `NOT_FOUND` to the `return_code`.

The line `case NOT_FOUND` is functionally equivalent to `NOT_FOUND = return_code`.

If `NOT_FOUND` had been an integer, the result would have been different:

```
>>> return_code = 200
... match return_code:
...     case 200:
...         print("200 found!")
...     case NOT_FOUND:
...         print("return code not found")
...
200 found!
>>> NOT_FOUND
Traceback (most recent call last):
  Python Shell, prompt 2, line 1
builtins.NameError: name 'NOT_FOUND' is not defined
```

Here the first case statement executes because `200 == 200`. Thus, `NOT_FOUND` is never set or initialized.

Quiz 109 - Unruly Unpacking

Unpacking two or more variables is a very common task in Python. You'll see it used in functions and methods often.

You'll also see it done with variable unpacking. You may even remember doing this in a previous quiz or two.

In this quiz you will attempt to "unpack" multiple strings into one or more variables.

How many get unpacked, and what is the data type at the end?

Do your best!

The Quiz

What is the output of this code?

```python
author = "Mike", "Driscoll"
*copy, = author
print(copy)
```

A. ['Mike']
B. ['Mike', 'Driscoll']
C. 'Mike', 'Driscoll'
D. 'Mike'

Hint

Looking up the term "unpacking generalizations" can help you find other examples on the Internet.

Answer 109 - Unruly Unpacking

This is the answer to **Quiz 109 - Unruly Unpacking**.

Your original question was:

What is the output of this code?

```python
author = "Mike", "Driscoll"
*copy, = author
print(copy)
```

A. [‘Mike’]
B. [‘Mike’, ‘Driscoll’]
C. ‘Mike’, ‘Driscoll’
D. ‘Mike’

The Answer

B) [‘Mike’, ‘Driscoll’]

Explanation

When you unpack multiple values from right to left, the left-hand side needs one of the following:

- The same number of variables on the left as the number of values on the right
- One variable with an asterisk attached to take in the extra values if there are not enough variables

In this quiz, you only have ONE variable for two values. To make that work, you need to add the asterisk, but you also need the comma!

You can see why in the following REPL session:

```
>>> author = "Mike", "Driscoll"

# Attempting to unpack with asterisk but no comma
>>> *copy = author
Traceback (most recent call last):
  Python Shell, prompt 6, line 1
Syntax Error: starred assignment target must be in a list or tuple: <string>, line 1\
, pos 1

# Unpacking with asterisk plus the comma
>>> *copy, = author
>>> print(copy)
['Mike', 'Driscoll']
```

Without the comma, you get a Syntax Error because the assignment target needs to be a list or tuple.

You can create a tuple by adding a comma at the end. Then Python will happily unpack the items for you!

Quiz 110 - Sorting Shutout

Data is often messy and needs to be cleaned or pre-processed. One common pre-processing task is sorting your data to make later cleaning procedures easier.

Python lists have a built-in `sort()` method that you can use.

The question here is what does the `sort()` method do?

If you know the answer, then you will make quick work of this quiz!

The Quiz

What is the output of this code?

```python
my_list = [3, 1, 10, 5]
my_list = my_list.sort()
print(my_list)
```

A. [1, 3, 5, 10]
B. []
C. [10, 5, 3, 1]
D. None

Hint

You may find a clue in the Python documentation[82]

[82]https://docs.python.org/3/tutorial/datastructures.html#more-on-lists

Answer 110 - Sorting Shutout

This is the answer to **Quiz 110 - Sorting Shutout**.

Your original question was:

What is the output of this code?

```python
my_list = [3, 1, 10, 5]
my_list = my_list.sort()
print(my_list)
```

A. [1, 3, 5, 10]
B. []
C. [10, 5, 3, 1]
D. None

The Answer

D) None

Explanation

This quiz highlights a common pitfall for beginners and occasionally even seasoned professionals. The list's `sort()` method sorts the list in place, which is useful when you don't want or need to create a new list.

However, that also means that `sort()` returns `None`.

If you want a new list returned, you can get that by using the `sorted()` function, which is one of Python's many built-in functions.

Here's an example:

```
>>> x = [3, 1, 10, 5]
>>> id(x)
4578669568
>>> x = sorted(x)
>>> id(x)
4578664320
```

The code above demonstrates that the list's id changes after sorting it. The reason is simple. The `sorted()` function creates a brand new list.

The `sort()` method is very handy. Just don't let it trip you up!

Afterword

I have always enjoyed puzzles and brain teasers. I am curious if there are enough like-minded people around who will buy this book, but it was refreshing to write. I have yet to write multiple chapters in a single day with any of my previous books, but I could sometimes write six or even eight with this quiz book!

I appreciate every one of you who did read this book, though. I look forward to hearing your thoughts and whether or not this book was valuable to you.

I hope you'll check out some of my previous works, especially my newer books, which I feel are much better than the first couple I wrote.

If you had problems with answering a lot of the questions in this book, you might consider purchasing Python 101: 2nd Edition:

- Amazon[83]
- Gumroad[84]
- Leanpub[85]

Or get a subscription at Teach Me Python[86].

Thanks again for all your support!

Mike

[83]https://www.amazon.com/Python-101-2nd-Michael-Driscoll-ebook/dp/B08GZTFCRF
[84]https://driscollis.gumroad.com/l/pypy101
[85]https://leanpub.com/py101
[86]https://www.teachmepython.com/