FullStack.Cafe - Tech Interview Plan

Q1: What are the built-in types available In Python? ☆

Topics: Python

Answer:

Immutable built-in datatypes of Python

- Numbers
- Strings
- Tuples

Mutable built-in datatypes of Python

- List
- Dictionaries
- Sets

Q2: Name some characteristics of Python? ☆

Topics: Python

Answer:

Here are a few key points:

- Python is an **interpreted language**. That means that, unlike languages like C and its variants, Python does not need to be compiled before it is run. Other interpreted languages include *PHP* and *Ruby*.
- Python is **dynamically typed**, this means that you don't need to state the types of variables when you declare them or anything like that. You can do things like `x=111` and then `x="I'm a string"` without error
- Python is well suited to **object orientated programming** in that it allows the definition of classes along with composition and inheritance. Python does not have access specifiers (like C++'s `public`, `private`), the justification for this point is given as "we are all adults here"
- In Python, **functions are first-class objects**. This means that they can be assigned to variables, returned from other functions and passed into functions. Classes are also first class objects
- Writing Python code is quick but running it is **often slower than compiled languages**. Fortunately, Python allows the inclusion of C based extensions so bottlenecks can be optimised away and often are. The `numpy` package is a good example of this, it's really quite quick because a lot of the number crunching it does isn't actually done by Python

Q3: How do I modify a string? ☆

Topics: Python

Answer:

You can’t, because strings are immutable. In most situations, you should simply construct a new string from the various parts you want to assemble it from.

Q4: Name some benefits of Python ☆☆

Topics: Python

Answer:

- Python is a dynamic-typed language. It means that you don’t need to mention the data type of variables during their declaration.
- Python supports object orientated programming as you can define classes along with the composition and inheritance.
- Functions in Python are like first-class objects. It suggests you can assign them to variables, return from other methods and pass as arguments.
- Developing using Python is quick but running it is often slower than compiled languages.
- Python has several usages like web-based applications, test automation, data modeling, big data analytics and much more.

Q5: What is PEP 8? ☆☆

Topics: Python

Answer:

PEP 8 is the latest Python coding standard, a set of coding recommendations. It guides to deliver more readable Python code.

Q6: Why would you use the "pass" statement? ☆☆

Topics: Python

Answer:

Python has the syntactical requirement that code blocks cannot be empty. Empty code blocks are however useful in a variety of different contexts, for example if you are designing a new class with some methods that you don't want to implement:

```
class MyClass(object):
    def meth_a(self):
        pass

    def meth_b(self):
        print "I'm meth_b"
```

If you were to leave out the pass, the code wouldn't run and you'll get an error:

```
IndentationError: expected an indented block
```

Other examples when we could use `pass`:

- Ignoring (all or) a certain type of `Exception`
- Deriving an exception class that does not add new behaviour
- Testing that code runs properly for a few test values, without caring about the results

Q7: What is lambda functions in Python? ☆☆

Topics: Python

Answer:

A **lambda function** is a small anonymous function. A lambda function can take any number of arguments, but can only have one expression.

Consider:

```
x = lambda a : a + 10
print(x(5)) # Output: 15
```

Q8: When to use a tuple vs list vs dictionary in Python? ☆☆

Topics: Python

Answer:

- Use a `tuple` to store a sequence of items that will not change.
- Use a `list` to store a sequence of items that may change.
- Use a `dictionary` when you want to associate pairs of two items.

Q9: What is pickling and unpickling? ☆☆

Topics: Python

Answer:

The pickle module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure.

- **Pickling** - is the process whereby a Python object hierarchy is converted into a byte stream,
- **Unpickling** - is the inverse operation, whereby a byte stream is converted back into an object hierarchy.

Q10: What is negative index in Python? ☆☆

Topics: Python

Answer:

Python sequences can be index in positive and negative numbers. For positive index, 0 is the first index, 1 is the second index and so forth. For negative index, (-1) is the last index and (-2) is the second last index and so forth.

Q11: What are local variables and global variables in Python? ☆☆

Topics: Python

Answer:

- **Global Variables**: Variables declared outside a function or in global space are called global variables. These variables can be accessed by any function in the program.

- **Local Variables**: Any variable declared inside a function is known as a local variable. This variable is present in the local space and not in the global space.

Q12: What are the rules for local and global variables in Python? ☆☆

Topics: Python

Answer:

In Python, variables that are only referenced inside a function are implicitly global. If a variable is assigned a value anywhere within the function's body, it's assumed to be a local unless explicitly declared as global.

Requiring global for assigned variables provides a bar against unintended side-effects.

Q13: What are some drawbacks of the Python language? ☆☆

Topics: Python

Answer:

The two most common valid answers to this question (by no means intended as an exhaustive list) are:

- The Global Interpreter Lock (GIL). CPython (the most common Python implementation) is not fully thread safe. In order to support multi-threaded Python programs, CPython provides a global lock that must be held by the current thread before it can safely access Python objects. As a result, no matter how many threads or processors are present, only one thread is ever being executed at any given time. In comparison, it is worth noting that the PyPy implementation discussed earlier in this article provides a stackless mode that supports micro-threads for massive concurrency.
- Execution speed. Python can be slower than compiled languages since it is interpreted. (Well, sort of. See our earlier discussion on this topic.)

Q14: Does Python have a switch-case statement? ☆☆

Topics: Python

Answer:

In Python, we do not have a switch-case statement. Here, you may write a switch function to use. Else, you may use a set of if-elif-else statements. To implement a function for this, we may use a dictionary.

```
def switch_demo(argument):
    switcher = {
        1: "January",
        2: "February",
        3: "March",
        4: "April",
        5: "May",
        6: "June",
        7: "July",
        8: "August",
        9: "September",
        10: "October",
```

```
        11: "November",
        12: "December"
    }
    print switcher.get(argument, "Invalid month")
```

Q15: What Is The Benefit Of Using Flask? ☆☆

Topics: Python

Answer:

Flask is part of the micro-framework. Which means it will have little to no dependencies on external libraries. It makes the framework light while there is little dependency to update and less security bugs.

Q16: Suppose lst is [2, 33, 222, 14, 25], What is lst[-1]? ☆☆

Topics: Python

Problem:

Suppose `lst` is `[2, 33, 222, 14, 25]`, What is `lst[-1]`?

Solution:

It's `25`. Negative numbers mean that you count from the right instead of the left. So, `lst[-1]` refers to the last element, `lst[-2]` is the second-last, and so on.

Q17: Given variables a and b, switch their values so that b has the value of a, and a has the value of b without using an intermediary variable. ☆☆

Topics: Python

Answer:

```
a, b = b, a
```

Q18: How do you list the functions in a module? ☆☆

Topics: Python

Answer:

Use the `dir()` method to list the functions in a module:

```
import some_module
print dir(some_module)
```

Q19: What are descriptors? ☆☆

Topics: Python

Answer:

Descriptors were introduced to Python way back in version 2.2. They provide the developer with the ability to add managed attributes to objects. The methods needed to create a descriptor are `__get__`, `__set__` and `__delete__`. If you define any of these methods, then you have created a descriptor.

Descriptors power a lot of the magic of Python's internals. They are what make properties, methods and even the super function work. They are also used to implement the new style classes that were also introduced in Python 2.2.

Q20: How to make a flat list out of list of lists? ☆☆☆

Topics: Python

Answer:

Given a list of lists `l` consider:

```
flat_list = []
for sublist in l:
    for item in sublist:
        flat_list.append(item)
```

Or using *lambda function*:

```
flatten = lambda l: [item for sublist in l for item in sublist]
print(flatten([[1],[2],[3],[4,5]]))
# Output: [1, 2, 3, 4, 5]
```

FullStack.Cafe - Tech Interview Plan

Q1: What's the difference between lists and tuples? ☆☆☆

Topics: Python

Answer:

The key difference is that `tuples` are immutable. This means that you cannot change the values in a tuple once you have created it. So if you're going to need to change the values use a `List`.

Apart from tuples being immutable there is also a semantic distinction that should guide their usage. Tuples are heterogeneous data structures (i.e., their entries have different meanings), while lists are homogeneous sequences. Tuples have structure, lists have order.

One example of tuple be pairs of page and line number to reference locations in a book, e.g.:

```
my_location = (42, 11)  # page number, line number
```

You can then use this as a key in a dictionary to store notes on locations. A list on the other hand could be used to store multiple locations. Naturally one might want to add or remove locations from the list, so it makes sense that lists are mutable. On the other hand it doesn't make sense to add or remove items from an existing location - hence tuples are immutable.

Q2: Is it possible to have static methods in Python? ☆☆☆

Topics: Python

Answer:

Use the `@staticmethod` decorator:

```
class MyClass(object):
    @staticmethod
    def the_static_method(x):
        print x

MyClass.the_static_method(2) # outputs 2
```

Q3: What are decorators in Python? ☆☆☆

Topics: Python

Answer:

In Python, functions are the first class objects, which means that:

- Functions are objects; they can be referenced to, passed to a variable and returned from other functions as well.
- Functions can be defined inside another function and can also be passed as argument to another function.

Decorators are very powerful and useful tool in Python since it allows programmers to modify the behavior of function or class. Decorators allow us to wrap another function in order to extend the behavior of wrapped function, without permanently modifying it.

```
@gfg_decorator
def hello_decorator():
    print("Gfg")

'''Above code is equivalent to:

def hello_decorator():
    print("Gfg")

hello_decorator = gfg_decorator(hello_decorator)'''
```

Q4: Explain how does Python memory management work? ☆☆☆

Topics: Python

Answer:

Python -- like C#, Java and many other languages -- uses *garbage collection* rather than manual memory management. You just freely create objects and the language's memory manager periodically (or when you specifically direct it to) looks for any objects that are no longer referenced by your program.

If you want to hold on to an object, just hold a reference to it. If you want the object to be freed (eventually) remove any references to it.

```
def foo(names):
  for name in names:
    print name

foo(["Eric", "Ernie", "Bert"])
foo(["Guthtrie", "Eddie", "Al"])
```

Each of these calls to foo creates a Python `list` object initialized with three values. For the duration of the foo call they are referenced by the variable names, but as soon as that function exits no variable is holding a reference to them and they are fair game for the garbage collector to delete.

Q5: What's the difference between the list methods append() and extend()? ☆☆☆

Topics: Python

Answer:

`append` *adds* an element to a list, and `extend` *concatenates* the first list with another list (or another iterable, not necessarily a list).

Consider:

```
x = [1, 2, 3]
x.append([4, 5])
print (x)
# [1, 2, 3, [4, 5]]
```

```
x = [1, 2, 3]
x.extend([4, 5])
print (x)
# [1, 2, 3, 4, 5]
```

Q6: How do I check if a list is empty? ☆☆☆

Topics: Python

Answer:

Based on PEP 8 style guide, for sequences (strings, lists, tuples), use the fact that empty sequences are *false*:

```
if not list:
  print("List is empty")
```

My problem with `if not list: ...` is that it gives the false impression that li is a boolean variable.

So more clear, explicit way would be:

```
if len(list) == 0:
    print('the list is empty')
```

Q7: What is namespace in Python? ☆☆☆

Topics: Python

Answer:

In Python, every name introduced has a place where it lives and can be hooked for. This is known as **namespace**. It is like a box where a variable name is mapped to the object placed. Whenever the variable is searched out, this box will be searched, to get corresponding object.

Q8: How can I create a copy of an object in Python? ☆☆☆

Topics: Python

Answer:

- To get a fully independent copy (deep copy) of an object you can use the `copy.deepcopy()` function. Deep copies are recursive copies of each interior object.
- For shallow copy use `copy.copy()`. Shallow copies are just copies of the outermost container.

Q9: What is the difference between range and xrange functions in Python 2.X? ☆☆☆

Topics: Python

Answer:

In Python 2.x:

- `range` creates a list, so if you do `range(1, 10000000)` it creates a list in memory with `9999999` elements.
- `xrange` is a generator object that evaluates lazily.

In Python 3, `range` does the equivalent of python's `xrange`, and to get the list, you have to use `list(range(...))`.

`xrange` brings you two advantages:

- You can iterate longer lists without getting a MemoryError.
- As it resolves each number lazily, if you stop iteration early, you won't waste time creating the whole list

Q10: How can you share global variables across modules? ☆☆☆

Topics: Python

Answer:

The canonical way to share information across modules within a single program is to create a special configuration module (often called `config` or `cfg`). Just import the configuration module in all modules of your application; the module then becomes available as a global name. Because there is only one instance of each module, any changes made to the module object get reflected everywhere.

File: config.py

```
x = 0 # Default value of the 'x' configuration setting
```

File: mod.py

```
import config
config.x = 1
```

File: main.py

```
import config
import mod
print config.x
```

Module variables are also often used to implement the Singleton design pattern, for the same reason.

Q11: What is monkey patching and is it ever a good idea? ☆☆☆

Topics: Python

Answer:

Monkey patching is changing the behaviour of a function or object after it has already been defined. For example:

```
import datetime
datetime.datetime.now = lambda: datetime.datetime(2012, 12, 12)
```

Most of the time it's a pretty terrible idea - it is usually best if things act in a well-defined way. One reason to monkey patch would be in testing. The mock package is very useful to this end.

Q12: What does this stuff mean: *args, **kwargs? And why would we use it? ☆☆☆

Topics: Python

Answer:

- Use `*args` when we aren't sure how many arguments are going to be passed to a function, or if we want to pass a stored list or tuple of arguments to a function.
- `**kwargs` is used when we dont know how many keyword arguments will be passed to a function, or it can be used to pass the values of a dictionary as keyword arguments.

Q13: Explain the UnboundLocalError exception and how to avoid it? ☆☆☆

Topics: Python

Problem:

Consider:

```
>>> x = 10
>>> def foo():
...     print(x)
...     x += 1
```

And the output:

```
>>> foo()
Traceback (most recent call last):
  ...
UnboundLocalError: local variable 'x' referenced before assignment
```

Why am I getting an UnboundLocalError when the variable has a value?

Solution:

When you make an assignment to a variable in a scope, that variable becomes local to that scope and shadows any similarly named variable in the outer scope. Since the last statement in foo assigns a new value to x, the compiler recognizes it as a local variable. Consequently when the earlier print(x) attempts to print the uninitialized local variable and an error results.

In the example above you can access the outer scope variable by declaring it `global`:

```
>>> x = 10
>>> def foobar():
...     global x
...     print(x)
...     x += 1
>>> foobar()
10
```

Q14: What is a None value? ☆☆☆

Topics: Python

Answer:

`None` is just a value that commonly is used to signify 'empty', or 'no value here'.

If you write a function, and that function doesn't use an explicit return statement, `None` is returned instead, for example.

Another example is to use `None` for default values. it is good programming practice to not use mutable objects as default values. Instead, use `None` as the default value and inside the function, check if the parameter is None and create a new list/dictionary/whatever if it is.

Consider:

```
def foo(mydict=None):
    if mydict is None:
        mydict = {}  # create a new dict for local namespace
```

Q15: What are immutable objects in Python? ☆☆☆

Topics: Python

Answer:

An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

Q16: What is the most efficient way to concatenate many strings together? ☆☆☆

Topics: Python

Answer:

`str` and `bytes` objects are immutable, therefore concatenating many strings together is inefficient as each concatenation creates a new object. In the general case, the total runtime cost is quadratic in the total string length.

To accumulate many `str` objects, I would recommend to place them into a list and call `str.join()` at the end:

```
chunks = []
for s in my_strings:
    chunks.append(s)
result = ''.join(chunks)
```

Q17: What are the key differences between Python 2 and 3? ☆☆☆

Topics: Python

Answer:

Here are some of the key differences that a developer should be aware of:

- Text and Data instead of Unicode and 8-bit strings. Python 3.0 uses the concepts of text and (binary) data instead of Unicode strings and 8-bit strings. The biggest ramification of this is that any attempt to mix text and data in Python 3.0 raises a TypeError (to combine the two safely, you must decode bytes or encode Unicode, but you need to know the proper encoding, e.g. UTF-8)

- This addresses a longstanding pitfall for naïve Python programmers. In Python 2, mixing Unicode and 8-bit data would work if the string happened to contain only 7-bit (ASCII) bytes, but you would get UnicodeDecodeError if it contained non-ASCII values. Moreover, the exception would happen at the combination point, not at the point at which the non-ASCII characters were put into the str object. This behavior was a common source of confusion and consternation for neophyte Python programmers.

- `print` function. The print statement has been replaced with a print() function

- `xrange` – buh-bye. xrange() no longer exists (range() now behaves like xrange() used to behave, except it works with values of arbitrary size)

- API changes:
  - `zip()`, `map()` and `filter()` all now return iterators instead of lists.
  - `dict.keys()`, `dict.items()` and `dict.values()` now return 'views' instead of lists.
  - `dict.iterkeys()`, `dict.iteritems()` and `dict.itervalues()` are no longer supported.
  - Comparison operators. The ordering comparison operators (<, <=, >=, >) now raise a TypeError exception when the operands don't have a meaningful natural ordering. Some examples of the ramifications of this include:
  - Expressions like 1 < '', 0 > None or len <= len are no longer valid
  - None < None now raises a TypeError instead of returning False
  - Sorting a heterogeneous list no longer makes sense.
  - All the elements must be comparable to each other

Q18: What Is Flask-WTF And What Are Their Features? ☆☆☆

Topics: Python

Answer:

Flask-WTF offers simple integration with WTForms. Features include for Flask-WTF are:

- Integration with wtforms
- Secure form with csrf token
- Global csrf protection
- Internationalization integration
- Recaptcha supporting

- File upload that works with Flask Uploads

Q19: Is this valid in Python and why? ☆☆☆

Topics: Python

Problem:

Consider:

```
def my_function():
  print my_function.what
my_function.what = "right?"
my_function() # Prints "right?"
```

Solution:

It is **valid**. In Python, everything is an object, including functions. But if we don't have what defined, we will get an Attribute error.

Q20: Write a program to check whether the object is of a class or its subclass. ☆☆☆

Topics: Python

Answer:

There is a method which is built-in to show the instances of an object that consists of many classes by providing a tuple in a table instead of individual classes. The method is `isinstance(obj,cls)`

`isinstance(obj, (class1, class2, ...))` is used to check the object's presence in one of the classes. The built in types can also have many formats of the same function like `isinstance(obj, str)` or `isinstance(obj, (int, long, float, complex))`:

```
def search(obj):
  if isinstance(obj, box):
  # This is the code that is given for the box and write the program in the object
  elif isinstance(obj, Document):
  # This is the code that searches the document and writes the values in it
  elif

obj.search()
#This is the function used to search the object's class.
```

FullStack.Cafe - Tech Interview Plan

Q1: What is the difference between range and xrange? How has this changed over time? ☆☆☆

Topics: Python

Answer:

As follows:

- `xrange` returns the xrange object while range returns the list, and uses the same memory and no matter what the range size is.
- For the most part, xrange and range are the exact same in terms of functionality. They both provide a way to generate a list of integers for you to use, however you please.
- The only difference is that range returns a Python list object and x range returns an xrange object. This means that xrange doesn't actually generate a static list at run-time like range does. It creates the values as you need them with a special technique called yielding. This technique is used with a type of object known as generators. That means that if you have a really gigantic range you'd like to generate a list for, say one billion, xrange is the function to use.
- This is especially true if you have a really memory sensitive system such as a cell phone that you are working with, as range will use as much memory as it can to create your array of integers, which can result in a Memory Error and crash your program. It's a memory hungry beast.

Q2: What is a "callable"? ☆☆☆

Topics: Python

Answer:

A **callable** is an object we can call - function or an object implementing the `__call__` special method. Any object can be made callable.

Q3: What is introspection/reflection and does Python support it? ☆☆☆

Topics: Python

Answer:

Introspection is the ability to examine an object at runtime. Python has a `dir()` function that supports examining the attributes of an object, `type()` to check the object type, `isinstance()`, etc.

While introspection is passive examination of the objects, reflection is a more powerful tool where we could modify objects at runtime and access them dynamically. E.g.

- `setattr()` adds or modifies an object's attribute;
- `getattr()` gets the value of an attribute of an object.

It can even invoke functions dynamically:

```
getattr(my_obj, "my_func_name")()
```

Q4: What are the Dunder/Magic/Special methods in Python? Name a few. ☆☆☆

Topics: Python

Answer:

Dunder (derived from double underscore) methods are special/magic predefined methods in Python, with names that start and end with a double underscore. There's nothing really magical about them. Examples of these include:

- `__init__` - constructor
- `__str__`, `__repr__` - object representation (casting to string, printing)
- `__len__`, `__next__` ... - generators
- `__enter__`, `__exit__` - context managers
- `__eq__`, `__lt__`, `__gt__` - operator overloading

Q5: What are virtualenvs? ☆☆☆

Topics: Python

Answer:

A **virtualenv** is what Python developers call an isolated environment for development, running, debugging Python code. It is used to isolate a Python interpreter together with a set of libraries and settings. Together with pip, it allows us to develop, deploy and run multiple applications on a single host, each with their own version of the Python interpreter, and separate set of libraries.

Q6: What is the python “with” statement designed for? ☆☆☆

Topics: Python

Answer:

The `with` statement simplifies exception handling by encapsulating common preparation and cleanup tasks in so-called *context managers*.

For instance, the `open` statement is a context manager in itself, which lets you open a file, keep it open as long as the execution is in the context of the `with` statement where you used it, and close it as soon as you leave the context, no matter whether you have left it because of an exception or during regular control flow.

As a result you could do something like:

```
with open("foo.txt") as foo_file:
    data = foo_file.read()
```

OR

```
from contextlib import nested
with nested(A(), B(), C()) as(X, Y, Z):
```

```
    do_something()
```

OR (Python 3.1)

```
with open('data') as input_file, open('result', 'w') as output_file:
    for line in input_file:
        output_file.write(parse(line))
```

OR

```
lock = threading.Lock()
with lock: #Critical section of code
```

Q7: How the string does get converted to a number? ☆☆☆

Topics: Python

Answer:

- To convert the string into a number the built-in functions are used like `int()` constructor. It is a data type that is used like `int ('1') == 1`.
- `float()` is also used to show the number in the format as `float('1') = 1`.
- The number by default are interpreted as decimal and if it is represented by `int('0x1')` then it gives an error as `ValueError`. In this the `int(string,base)` function takes the parameter to convert string to number in this the process will be like `int('0x1',16) == 16`. If the base parameter is defined as 0 then it is indicated by an octal and 0x indicates it as hexadecimal number.
- There is function `eval()` that can be used to convert string into number but it is a bit slower and present many security risks

Q8: What is the function of "self"? ☆☆☆

Topics: Python

Answer:

Self is a variable that represents the instance of the object to itself. In most of the object oriented programming language, this is passed to the methods as a hidden parameters that is defined by an object. But, in python, it is declared and passed explicitly. It is the first argument that gets created in the instance of the class A and the parameters to the methods are passed automatically. It refers to separate instance of the variable for individual objects.

Let's say you have a class `ClassA` which contains a method `methodA` defined as:

```
def methodA(self, arg1, arg2): #do something
```

and `ObjectA` is an instance of this class.

Now when `ObjectA.methodA(arg1, arg2)` is called, python internally converts it for you as:

```
ClassA.methodA(ObjectA, arg1, arg2)
```

The `self` variable refers to the object itself.

Q9: What is the process of compilation and linking in Python? ☆☆☆

Topics: Python

Answer:

Compilation: The source code in python is saved as a .py file which is then compiled into a format known as byte code, byte code is then converted to machine code. After the compilation, the code is stored in .pyc files and is regenerated when the source is updated. This process is known as compilation.

Linking: Linking is the final phase where all the functions are linked with their definitions as the linker knows where all these functions are implemented. This process is known as linking.

Q10: What does the Python nonlocal statement do (in Python 3.0 and later)? ☆☆☆

Topics: Python

Answer:

In short, it lets you assign values to a variable in an outer (but non-global) scope.

The `nonlocal` statement causes the listed identifiers to refer to previously bound variables in the nearest enclosing scope excluding globals.

For example the counter generator can be rewritten to use this so that it looks more like the idioms of languages with closures.

```
def make_counter():
    count = 0
    def counter():
        nonlocal count
        count += 1
        return count
    return counter
```

Q11: After executing the above code, what is the value of y? ☆☆☆

Topics: Python

Problem:

```
>>> x = 100
>>> y = x
>>> x = 200
```

After executing the above code, what is the value of y?

Solution:

```
>>> print(y)
100
```

Assignment in Python means one thing, and one thing only: **The variable named on the left should now refer to the value on the right.**

In other words, when I said:

```
y = x
```

Python doesn't read this as, "y should now refer to the variable x." Rather, it read it as, "y should now refer to whatever value x refers to."

Because x refers to the integer 100, y now refers to the integer 100. After these two assignments ("x = 100" and "y = x"), there are now two references to the integer 100 that didn't previously exist.

When we say that "x = 200", we're removing one of those references, such that x no longer refers to 100. Instead, x will now refer to the integer 200.

Q12: What are the wheels and eggs? What is the difference? ☆☆☆

Topics: Python

Answer:

Wheel and **Egg** are both packaging formats that aim to support the use case of needing an install artifact that doesn't require building or compilation, which can be costly in testing and production workflows.

The Egg format was introduced by setuptools in 2004, whereas the Wheel format was introduced by PEP 427 in 2012.

Wheel is currently considered the standard for built and binary packaging for Python.

Here's a breakdown of the important differences between Wheel and Egg.

- Wheel has an official PEP. Egg did not.
- Wheel is a distribution format, i.e a packaging format. 1 Egg was both a distribution format and a runtime installation format (if left zipped), and was designed to be importable.
- Wheel archives do not include .pyc files. Therefore, when the distribution only contains Python files (i.e. no compiled extensions), and is compatible with Python 2 and 3, it's possible for a wheel to be "universal", similar to an sdist.
- Wheel uses PEP376-compliant .dist-info directories. Egg used .egg-info.
- Wheel has a richer file naming convention. A single wheel archive can indicate its compatibility with a number of Python language versions and implementations, ABIs, and system architectures.
- Wheel is versioned. Every wheel file contains the version of the wheel specification and the implementation that packaged it.
- Wheel is internally organized by sysconfig path type, therefore making it easier to convert to other formats.

Q13: What are metaclasses in Python? ☆☆☆☆

Topics: Python

Answer:

A **metaclass** is the class of a class. A class defines how an instance of the class (i.e. an object) behaves while a metaclass defines how a class behaves. A class is an instance of a metaclass. You can call it a 'class factory'.

Q14: How to make a chain of function decorators? ☆☆☆☆

Topics: Python

Problem:

How can I make two decorators in Python that would do the following?

```
@makebold
@makeitalic
def say():
   return "Hello"
```

which should return:

```
"<b><i>Hello</i></b>"
```

Solution:

Consider:

```
from functools import wraps

def makebold(fn):
    @wraps(fn)
    def wrapped(*args, **kwargs):
        return "<b>" + fn(*args, **kwargs) + "</b>"
    return wrapped

def makeitalic(fn):
    @wraps(fn)
    def wrapped(*args, **kwargs):
        return "<i>" + fn(*args, **kwargs) + "</i>"
    return wrapped

@makebold
@makeitalic
def hello():
    return "hello world"

@makebold
@makeitalic
def log(s):
    return s

print hello()        # returns "<b><i>hello world</i></b>"
print hello.__name__ # with functools.wraps() this returns "hello"
print log('hello')   # returns "<b><i>hello</i></b>"
```

Q15: What is the difference between @staticmethod and @classmethod? ☆☆☆☆

Topics: Python

Answer:

A **staticmethod** is a method that knows nothing about the class or instance it was called on. It just gets the arguments that were passed, no implicit first argument. It's definition is immutable via inheritance.

```
class C:
    @staticmethod
    def f(arg1, arg2, ...): ...
```

A **classmethod**, on the other hand, is a method that gets passed the class it was called on, or the class of the instance it was called on, as first argument. Its definition follows Sub class, not Parent class, via inheritance.

```
class C:
   @classmethod
   def f(cls, arg1, arg2, ...): ...
```

If your method accesses other variables/methods in your class then use `@classmethod`.

Q16: What's the difference between a Python module and a Python package? ☆☆☆☆

Topics: Python

Answer:

Any Python file is a **module**, its name being the file's base name without the .py extension.

```
import my_module
```

A **package** is a collection of Python modules: while a module is a single Python file, a package is a directory of Python modules containing an additional **init**.py file, to distinguish a package from a directory that just happens to contain a bunch of Python scripts. Packages can be nested to any depth, provided that the corresponding directories contain their own **init**.py file.

Packages are modules too. They are just packaged up differently; they are formed by the combination of a directory plus **init**.py file. They are modules that can contain other modules.

```
from my_package.timing.danger.internets import function_of_love
```

Q17: What is GIL? ☆☆☆☆

Topics: Python

Answer:

Python has a construct called the **Global Interpreter Lock** (GIL). The GIL makes sure that only one of your 'threads' can execute at any one time. A thread acquires the GIL, does a little work, then passes the GIL onto the next thread. This happens very quickly so to the human eye it may seem like your threads are executing in

parallel, but they are really just taking turns using the same CPU core. All this GIL passing adds overhead to execution.

Q18: Is there a tool to help find bugs or perform static analysis? ☆☆☆☆

Topics: Python

Answer:

PyChecker is a static analysis tool that finds bugs in Python source code and warns about code complexity and style.

Pylint is another tool that checks if a module satisfies a coding standard, and also makes it possible to write plug-ins to add a custom feature. In addition to the bug checking that PyChecker performs, Pylint offers some additional features such as checking line length, whether variable names are well-formed according to your coding standard, whether declared interfaces are fully implemented, and more.

Q19: Why are default values shared between objects? ☆☆☆☆

Topics: Python

Answer:

It is often expected that a function call creates new objects for *default values*. This is not what happens. Default values are created exactly once, when the function is defined. If that object is changed, like the dictionary in this example, subsequent calls to the function will refer to this changed object.

Consider:

```
def foo(mydict={}):  # Danger: shared reference to one dict for all calls
    ... compute something ...
    mydict[key] = value
    return mydict
```

The first time you call this function, mydict contains a single item. The second time, mydict contains two items because when `foo()` begins executing, mydict starts out with an item already in it.

Q20: How do I write a function with output parameters (call by reference) ☆☆☆☆

Topics: Python

Answer:

In Python arguments are passed *by assignment*. When you call a function with a parameter, a **new reference** is created that refers to the object passed in. This is separate from the reference that was used in the function call, so there's no way to update that reference and make it refer to a new object.

If you pass a *mutable* object into a method, the method gets a reference to that same object and you can mutate it to your heart's delight, but if you rebind the reference in the method (like `b = b + 1`), the outer scope will know nothing about it, and after you're done, the outer reference will still point at the original object.

So to achieve the desired effect your best choice is to return a tuple containing the multiple results:

```
def func2(a, b):
    a = 'new-value'        # a and b are local names
    b = b + 1              # assigned to new objects
    return a, b            # return new values

x, y = 'old-value', 99
x, y = func2(x, y)
print(x, y)                # output: new-value 100
```

FullStack.Cafe - Tech Interview Plan

Q1: Create function that similar to os.walk ☆☆☆☆

Topics: Python

Answer:

```
def print_directory_contents(sPath):
    import os
    for sChild in os.listdir(sPath):
        sChildPath = os.path.join(sPath,sChild)
        if os.path.isdir(sChildPath):
            print_directory_contents(sChildPath)
        else:
            print(sChildPath)
```

Q2: Is it a good idea to use multi-thread to speed your Python code? ☆☆☆☆

Topics: Python

Answer:

Python doesn't allow multi-threading in the truest sense of the word. It has a multi-threading package but if you want to multi-thread to speed your code up, then it's usually not a good idea to use it.

Python has a construct called the Global Interpreter Lock (GIL). The GIL makes sure that only one of your 'threads' can execute at any one time. A thread acquires the GIL, does a little work, then passes the GIL onto the next thread. This happens very quickly so to the human eye it may seem like your threads are executing in parallel, but they are really just taking turns using the same CPU core. All this GIL passing adds overhead to execution.

Q3: What will be returned by this code? ☆☆☆☆

Topics: Python

Problem:

Consider:

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda: x**2)
>>> squares[2]()
>>> squares[4]()
```

Solution:

It will return:

```
>>> squares[2]()
16
>>> squares[4]()
16
```

This happens because `x` is not local to the lambdas, but is defined in the outer scope, and it is accessed when the lambda is called — not when it is defined. At the end of the loop, the value of `x` is 4, so all the functions now return 4**2, i.e. 16.

Q4: How is memory managed in Python? ☆☆☆☆

Topics: Python

Answer:

Python memory is managed by Python private heap space. All Python objects and data structures are located in a private heap. The programmer does not have an access to this private heap and interpreter. Like other programming language python also has garbage collector which will take care of memory management in python.Python also have an inbuilt garbage collector, which recycle all the unused memory and frees the memory and makes it available to the heap space. The allocation of Python heap space for Python objects is done by Python memory manager. The core API gives access to some tools for the programmer to code.

Python has a private heap space to hold all objects and data structures. Being programmers, we cannot access it; it is the interpreter that manages it. But with the core API, we can access some tools. The Python memory manager controls the allocation.

Q5: Whenever you exit Python, is all memory de-allocated? ☆☆☆☆

Topics: Python

Answer:

The answer here is **no**. The modules with circular references to other objects, or to objects referenced from global namespaces, aren't always freed on exiting Python. Plus, it is impossible to de-allocate portions of memory reserved by the C library.

Q6: Why are Python's 'private' methods not actually private? ☆☆☆☆

Topics: Python

Problem:

Python gives us the ability to create 'private' methods and variables within a class by prepending double underscores to the name, like this: `__myPrivateMethod()`. How, then, can one explain this?

```
>>> class MyClass:
...     def myPublicMethod(self):
...             print 'public method'
...     def __myPrivateMethod(self):
...             print 'this is private!!'
...
>>> obj = MyClass()
>>> obj.myPublicMethod()
```

```
public method
>>> obj.__myPrivateMethod()
Traceback (most recent call last):
  File "", line 1, in
AttributeError: MyClass instance has no attribute '__myPrivateMethod'
>>> dir(obj)
['_MyClass__myPrivateMethod', '__doc__', '__module__', 'myPublicMethod']
>>> obj._MyClass__myPrivateMethod()
this is private!!
```

Solution:

In short, the **name scrambling** is used to ensure that subclasses don't accidentally override the private methods and attributes of their superclasses. It's not designed to prevent deliberate access from outside.

Strictly speaking, private methods are accessible outside their class, just not easily accessible. *Nothing in Python is truly private*; internally, the names of private methods and attributes are mangled and unmangled on the fly to make them seem inaccessible by their given names.

You can access the `__parse` method of the `MP3FileInfo` class by the name `_MP3FileInfo__parse`. Acknowledge that this is interesting, then promise to never, ever do it in real code. Private methods are private for a reason, but like many other things in Python, their privateness is ultimately a matter of convention, not force.

Q7: Show me three different ways of fetching every third item in the list ☆☆☆☆

Topics: Python

Answer:

```
[x for i, x in enumerate(thelist) if i%3 == 0]

for i, x in enumerate(thelist):
    if i % 3: continue
    yield x

a = 0
for x in thelist:
    if a%3: continue
    yield x
    a += 1
```

Q8: Can you explain closures (as they relate to Python)? ☆☆☆☆

Topics: Python

Answer:

Objects are data with methods attached, **closures** are functions with data attached. The method of binding data to a function without actually passing them as parameters is called **closure**.

```
def make_counter():
    i = 0
    def counter(): # counter() is a closure
        nonlocal i
        i += 1
        return i
```

```
    return counter

c1 = make_counter()
c2 = make_counter()

print (c1(), c1(), c2(), c2())
# -> 1 2 1 2
```

Q9: What will be the output of the code below? ☆☆☆☆

Topics: Python

Problem:

Consider:

```
list = ['a', 'b', 'c', 'd', 'e']
print list[10:]
```

Solution:

The above code will output `[]`, and will not result in an `IndexError`.

As one would expect, attempting to access a member of a list using an index that exceeds the number of members (e.g., attempting to access `list[10]` in the list above) results in an `IndexError`. However, attempting to access a slice of a list at a starting index that exceeds the number of members in the list will not result in an `IndexError` and will simply return an empty list.

What makes this a particularly nasty gotcha is that it can lead to bugs that are really hard to track down since no error is raised at runtime.

Q10: Will the code below work? Why or why not? ☆☆☆☆

Topics: Python

Problem:

Given the following subclass of dictionary:

```
class DefaultDict(dict):
  def __missing__(self, key):
    return []
```

Will the code below work? Why or why not?

```
d = DefaultDict()
d['florp'] = 127
```

Solution:

Actually, the code shown will work with the standard dictionary object in python 2 or 3—that is normal behavior. Subclassing dict is unnecessary. However, the subclass still won't work with the code shown because `__missing__` returns a value but does not change the dict itself:

```
d = DefaultDict()
print d
{}
print d['foo']
[]
print d
{}
```

So it will "work," in the sense that it won't produce any error, but doesn't do what it seems to be intended to do.

Here is a `__missing__`-based method that will update the dictionary, as well as return a value:

```
class DefaultDict(dict):
    def __missing__(self, key):
        newval = []
        self[key] = newval
        return newval
```

But since version 2.5, a defaultdict object has been available in the collections module (in the standard library.)

Q11: What is the purpose of the single underscore "_" variable in Python? ☆☆☆☆

Topics: Python

Answer:

`_` has 4 main conventional uses in Python:

1. To hold the result of the last executed expression(/statement) in an interactive interpreter session. This precedent was set by the standard CPython interpreter, and other interpreters have followed suit
2. For translation lookup in i18n (see the gettext documentation for example), as in code like: `raise forms.ValidationError(_("Please enter a correct username"))`
3. As a general purpose "throwaway" variable name to indicate that part of a function result is being deliberately ignored (Conceptually, it is being discarded.), as in code like: `label, has_label, _ = text.partition(':')`.
4. As part of a function definition (using either `def` or `lambda`), where the signature is fixed (e.g. by a callback or parent class API), but this particular function implementation doesn't need all of the parameters, as in code like: `callback = lambda _: True`

Q12: Explain how you reverse a generator? ☆☆☆☆

Topics: Python

Answer:

You cannot reverse a generator in any generic way except by casting it to a sequence and creating an iterator from that. Later terms of a generator cannot necessarily be known until the earlier ones have been calculated.

Even worse, you can't know if your generator will ever hit a StopIteration exception until you hit it, so there's no way to know what there will even be a first term in your sequence.

The best you could do would be to write a reversed_iterator function:

```
def reversed_iterator(iter):
    return reversed(list(iter))
```

Q13: How is set() implemented internally? ☆☆☆☆

Topics: Python

Problem:

I've seen people say that `set` objects in python have `O(1)` membership-checking. How are they implemented internally to allow this? What sort of data structure does it use? What other implications does that implementation have?

Solution:

Indeed, CPython's sets are implemented as something like dictionaries with dummy values (the keys being the members of the set), with some optimization(s) that exploit this lack of values.

So basically a **set** uses a **hashtable** as its underlying data structure. This explains the `O(1)` membership checking, since looking up an item in a hashtable is an `O(1)` operation, on average.

Also, it worth to mention when people say sets have `O(1)` membership-checking, they are talking about the average case. In the worst case (when all hashed values collide) membership-checking is `O(n)`.

Q14: What is MRO in Python? How does it work? ☆☆☆☆

Topics: Python

Answer:

Method Resolution Order (MRO) it denotes the way a programming language resolves a method or attribute. Python supports classes inheriting from other classes. The class being inherited is called the Parent or Superclass, while the class that inherits is called the Child or Subclass.

In Python,** method resolution order** *defines the order in which the base classes are searched when executing a method*. First, the method or attribute is searched within a class and then it follows the order we specified while inheriting. This order is also called Linearization of a class and set of rules are called MRO (Method Resolution Order). While inheriting from another class, the interpreter needs a way to resolve the methods that are being called via an instance. Thus we need the method resolution order.

Python resolves method and attribute lookups using the C3 linearisation of the class and its parents. The C3 linearisation is **neither depth-first nor breadth-first** in complex multiple inheritance hierarchies.

Q15: What does an 'x = y or z' assignment do in Python? ☆☆☆☆

Topics: Python

Answer:

```
x = a or b
```

If `bool(a)` returns `False`, then `x` is assigned the value of `b`.

Q16: What is the difference between old style and new style classes in Python? ☆☆☆☆

Topics: Python

Problem:

What is the difference between old style and new style classes in Python? When should I use one or the other?

Solution:

Declaration-wise:

New-style classes inherit from object, or from another new-style class.

```
class NewStyleClass(object):
    pass

class AnotherNewStyleClass(NewStyleClass):
    pass
```

Old-style classes don't.

```
class OldStyleClass():
    pass
```

Python 3 Note:

Python 3 doesn't support old style classes, so either form noted above results in a new-style class.

Also, **MRO (Method Resolution Order)** changed:

- Classic classes do a depth first search from left to right. Stop on first match. They do not have the **mro** attribute.
- New-style classes MRO is more complicated to synthesize in a single English sentence. One of its properties is that a Base class is only searched for once all its Derived classes have been. They have the **mro** attribute which shows the search order.

Some other notes:

- New style class objects cannot be raised unless derived from `Exception`.
- Old style classes are still marginally faster for attribute lookup.

Q17: Why Python (CPython and others) uses the GIL? ☆☆☆☆

Topics: Python

Answer:

In CPython, the global interpreter lock, or GIL, is a mutex that prevents multiple native threads from executing Python bytecodes at once. This lock is necessary mainly because CPython's memory management is not thread-safe.

Python has a GIL as opposed to fine-grained locking for several reasons:

- It is faster in the single-threaded case.
- It is faster in the multi-threaded case for i/o bound programs.
- It is faster in the multi-threaded case for cpu-bound programs that do their compute-intensive work in C libraries.
- It makes C extensions easier to write: there will be no switch of Python threads except where you allow it to happen (i.e. between the Py_BEGIN_ALLOW_THREADS and Py_END_ALLOW_THREADS macros).
- It makes wrapping C libraries easier. You don't have to worry about thread-safety. If the library is not thread-safe, you simply keep the GIL locked while you call it.

Q18: What is an alternative to GIL? ☆☆☆☆

Topics: Python

Answer:

The Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there's a global lock, called the global interpreter lock or GIL, that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.

If the purpose of the GIL is to protect state from corruption, then one obvious alternative is **lock at a much finer grain (fine-grained locking)**; perhaps at a per object level. The problem with this is that although it has been demonstrated to increase the performance of multi-threaded programs, it has more overhead and single-threaded programs suffer as a result.

Q19: How to work with transitive dependencies? ☆☆☆☆

Topics: Python

Answer:

Transitive dependency is expressing the dependency of A on C when A depends on B and B depends on C.

If a transitive dependency is not explicitly specified in a project's `requirements.txt`, pip will grab the version of the required library specified in the project's `install_requires` section (of `setup.py`). If this section does not explicitly pin a version, you end up getting the latest version of that library.

If your application needs a specific version of a transitive dependency, pin it yourself in your application's `requirements.txt` file. Then pip will do the right thing.

Q20: What is Cython? ☆☆☆☆

Topics: Python

Answer:

Cython is a programming language that aims to be a superset of the Python programming language, designed to give C-like performance with code that is written mostly in Python with optional additional C-inspired syntax. Cython is a compiled language that is typically used to generate CPython extension modules.