Python Interview Questions & Answers

Q 1: What is Python?

Ans: Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. Python combines remarkable power with very clear syntax. It has interfaces to many system calls and libraries, as well as to various window systems, and is extensible in C or C++. It is also usable as an extension language for applications that need a programmable interface. Finally, Python is portable: it runs on many Unix variants, on the Mac, and on PCs under MS-DOS, Windows, Windows NT, and OS/2.

Q 2: Why can't I use an assignment in an expression?

Ans: Many people used to C or Perl complain that they want to use this C idiom:

```
while (line = readline(f)) {
...do something with line...
}
```

where in Python you're forced to write this:

```
while True:
line = f.readline() if not line:
break
...do something with line...
```

The reason for not allowing assignment in Python expressions is a common, hard-to-find bug in those other languages, caused by this construct:

```
if (x = 0) {
...error handling...
}
else {
```

...code that only works for nonzero x...

}

The error is a simple typo: x = 0, which assigns 0 to the variable x, was written while the comparison x == 0 is certainly what was intended.

Many alternatives have been proposed. Most are hacks that save some typing but use arbitrary or cryptic syntax or keywords, and fail the simple criterion for language change proposals: it should intuitively suggest the proper meaning to a human reader who has not yet been introduced to the construct.

Q 3: How do you set a global variable in a function?

Ans: x = 1 # make a global

def f():

print x # try to print the global

for j in range(100):

if q>3:

x=4

Any variable assigned in a function is local to that function. unless it is specifically declared global. Since a value is bound to x as the last statement of the function body, the compiler assumes that x is local. Consequently the print x attempts to print an uninitialized local variable and will trigger a NameError.

The solution is to insert an explicit global declaration at the start of the function:

def f():

global x

print x # try to print the global

for j in range(100):

if q>3:

x=4

In this case, all references to x are interpreted as references to the x from the module namespace.

Q 4: What are the rules for local and global variables in Python?

Ans: In Python, variables that are only referenced inside a function are implicitly global. If a variable is assigned a new value anywhere within the function's body, it's assumed to be a local. If a variable is ever assigned a new value inside the function, the variable is implicitly local, and you need to explicitly declare it as 'global'.

Though a bit surprising at first, a moment's consideration explains this. On one hand, requiring global for assigned variables provides a bar against unintended side-effects. On the other hand, if global was required for all global references, you'd be using global all the time. You'd have to declare as global every reference to a built-in function or to a component of an imported module. This clutter would defeat the usefulness of the global declaration for identifying side-effects.

Q 5: How do I share global variables across modules?

Ans: The canonical way to share information across modules within a single program is to create a special module (often called config or cfg). Just import the config 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.

For example:

config.py:

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

mod.py:

import config

config.x = 1

main.py:

import config

import mod

print config.x

Note that using a module is also the basis for implementing the Singleton design pattern, for the same reason.

Q 6: How can I pass optional or keyword parameters from one function to another?

Ans: Collect the arguments using the * and ** specifier in the function's parameter list; this gives you the positional arguments as a tuple and the keyword arguments as a dictionary. You can then pass these arguments when calling another function by using * and **:

def f(x, *tup, **kwargs):

kwargs['width']='14.3c'

g(x, *tup, **kwargs)

In the unlikely case that you care about Python versions older than 2.0, use 'apply': def f(x, *tup, **kwargs):

kwargs['width']='14.3c'

apply(g, (x,)+tup, kwargs)

Q 7: How do you make a higher order function in Python?

Ans: You have two choices: you can use nested scopes or you can use callable objects. For example,

suppose you wanted to define linear(a,b) which returns a function f(x) that computes the value a*x+b. Using nested scopes:

def linear(a,b):

def result(x):

return a*x + b

return result

Or using a callable object:

class linear:

def __init__(self, a, b):

self.a, self.b = a,b

def __call__(self, x):

return self.a * x + self.b

In both cases:

taxes = linear(0.3,2)

gives a callable object where taxes(10e6) == 0.3 * 10e6 + 2.

The callable object approach has the disadvantage that it is a bit slower and results in slightly longer code. However, note that a collection of callables can share their signature via inheritance:

class exponential(linear):

__init__ inherited

def __call__(self, x):

return self.a * (x ** self.b)

Object can encapsulate state for several methods:

class counter:

value = 0

def set(self, x): self.value = x

def up(self): self.value=self.value+1

def down(self): self.value=self.value-1

count = counter()

inc, dec, reset = count.up, count.down, count.set

Here inc(), dec() and reset() act like functions which share the same counting variable.

Q 8: How do I copy an object in Python?

Ans: In general, try copy.copy() or copy.deepcopy() for the general case. Not all objects can be copied, but most can.

Some objects can be copied more easily. Dictionaries have a copy() method: newdict = olddict.copy()

Sequences can be copied by slicing: new_l = l[:]

Q 9: How can I find the methods or attributes of an object?

Ans: For an instance x of a user-defined class, dir(x) returns an alphabetized list of the names containing the instance attributes and methods and attributes defined by its class.

Q 10: How do I convert a string to a number?

Ans: For integers, use the built-in int() type constructor, e.g. int('144') == 144. Similarly, float() converts to floating-point, e.g. float('144') == 144.0.

By default, these interpret the number as decimal, so that int('0144') == 144 and int('0x144') raises ValueError. int(string, base) takes the base to convert from as a second optional argument, so int('0x144', 16) == 324. If the base is specified as 0, the number is interpreted using Python's rules: a leading '0' indicates octal, and '0x' indicates a hex number.

Do not use the built-in function eval() if all you need is to convert strings to numbers. eval() will be significantly slower and it presents a security risk: someone could pass you a Python expression that might have unwanted side effects. For example, someone could pass __import__('os').system("rm -rf $HOME") which would erase your home directory.

eval() also has the effect of interpreting numbers as Python expressions, so that e.g. eval('09') gives a syntax error because Python regards numbers starting with '0' as octal (base 8).

Q 11: How do I convert a number to a string?

Ans: To convert, e.g., the number 144 to the string \'144\', use the built-in function str(). If you want a hexadecimal or octal representation, use the built-in functions hex() or oct(). For fancy formatting, use the % operator on strings, e.g. \"%04d\" % 144 yields \'0144\' and \"%.3f\" % (1/3.0) yields \'0.333\'. See the library reference manual for details.