Programming FAQ

Contents

- Programming FAQ
  - General Questions
    - Is there a source code level debugger with breakpoints, single-stepping, etc.?
    - Are there tools to help find bugs or perform static analysis?
    - How can I create a stand-alone binary from a Python script?
    - Are there coding standards or a style guide for Python programs?
  - Core Language
    - Why am I getting an UnboundLocalError when the variable has a value?
    - What are the rules for local and global variables in Python?
    - Why do lambdas defined in a loop with different values all return the same result?
    - How do I share global variables across modules?
    - What are the “best practices” for using import in a module?
    - Why are default values shared between objects?
    - How can I pass optional or keyword parameters from one function to another?
    - What is the difference between arguments and parameters?
    - Why did changing list ‘y’ also change list ‘x’?
    - How do I write a function with output parameters (call by reference)?
    - How do you make a higher order function in Python?
    - How do I copy an object in Python?
    - How can I find the methods or attributes of an object?
    - How can my code discover the name of an object?
    - What’s up with the comma operator’s precedence?
    - Is there an equivalent of C’s “?:” ternary operator?
    - Is it possible to write obfuscated one-liners in Python?
    - What does the slash(/) in the parameter list of a function mean?
  - Numbers and strings
    - How do I specify hexadecimal and octal integers?
    - Why does -22 // 10 return -3?
    - How do I get int literal attribute instead of SyntaxError?
    - How do I convert a string to a number?
    - How do I convert a number to a string?
    - How do I modify a string in place?
    - How do I use strings to call functions/methods?
    - Is there an equivalent to Perl’s chomp() for removing trailing newlines from strings?
    - Is there a scanf() or sscanf() equivalent?
    - What does ‘UnicodeDecodeError’ or ‘UnicodeEncodeError’ error mean?
  - Performance
    - My program is too slow. How do I speed it up?
    - What is the most efficient way to concatenate many strings together?
  - Sequences (Tuples/Lists)
    - How do I convert between tuples and lists?
    - What’s a negative index?
    - How do I iterate over a sequence in reverse order?
    - How do you remove duplicates from a list?
    - How do you remove multiple items from a list
    - How do you make an array in Python?

- How do I apply a method to a sequence of objects?
- Why does a_tuple[i] += ['item'] raise an exception when the addition works?
- I want to do a complicated sort: can you do a Schwartzian Transform in Python?
- How can I sort one list by values from another list?

- Objects
  - What is a class?
  - What is a method?
  - What is self?
  - How do I check if an object is an instance of a given class or of a subclass of it?
  - What is delegation?
  - How do I call a method defined in a base class from a derived class that extends it?
  - How can I organize my code to make it easier to change the base class?
  - How do I create static class data and static class methods?
  - How can I overload constructors (or methods) in Python?
  - I try to use __spam and I get an error about _SomeClassName__spam.
  - My class defines __del__ but it is not called when I delete the object.
  - How do I get a list of all instances of a given class?
  - Why does the result of `id()` appear to be not unique?
  - When can I rely on identity tests with the *is* operator?
  - How can a subclass control what data is stored in an immutable instance?
  - How do I cache method calls?
- Modules
  - How do I create a .pyc file?
  - How do I find the current module name?
  - How can I have modules that mutually import each other?
  - __import__('x.y.z') returns <module 'x'>; how do I get z?
  - When I edit an imported module and reimport it, the changes don't show up. Why does this happen?

General Questions

Is there a source code level debugger with breakpoints, single-stepping, etc.?

Yes.

Several debuggers for Python are described below, and the built-in function `breakpoint()` allows you to drop into any of them.

The pdb module is a simple but adequate console-mode debugger for Python. It is part of the standard Python library, and is `documented in the Library Reference Manual`. You can also write your own debugger by using the code for pdb as an example.

The IDLE interactive development environment, which is part of the standard Python distribution (normally available as Tools/scripts/idle), includes a graphical debugger.

PythonWin is a Python IDE that includes a GUI debugger based on pdb. The PythonWin debugger colors breakpoints and has quite a few cool features such as debugging non-PythonWin programs. PythonWin is available as part of pywin32 project and as a part of the ActivePython distribution.

Eric is an IDE built on PyQt and the Scintilla editing component.

trepan3k is a gdb-like debugger.

Visual Studio Code is an IDE with debugging tools that integrates with version-control software

There are a number of commercial Python IDEs that include graphical debuggers. They include:

- Wing IDE
- Komodo IDE
- PyCharm

Are there tools to help find bugs or perform static analysis?

Yes.

Pylint and Pyflakes do basic checking that will help you catch bugs sooner.

Static type checkers such as Mypy, Pyre, and Pytype can check type hints in Python source code.

How can I create a stand-alone binary from a Python script?

You don't need the ability to compile Python to C code if all you want is a stand-alone program that users can download and run without having to install the Python distribution first. There are a number of tools that determine the set of modules required by a program and bind these modules together with a Python binary to produce a single executable.

One is to use the freeze tool, which is included in the Python source tree as `Tools/freeze`. It converts Python byte code to C arrays; with a C compiler you can embed all your modules into a new program, which is then linked with the standard Python modules.

It works by scanning your source recursively for import statements (in both forms) and looking for the modules in the standard Python path as well as in the source directory (for built-in modules). It then turns the bytecode for modules written in Python into C code (array initializers that can be turned into code objects using the marshal module) and creates a custom-made config file that only contains those built-in modules which are actually used in the program. It then compiles the generated C code and links it with the rest of the Python interpreter to form a self-contained binary which acts exactly like your script.

The following packages can help with the creation of console and GUI executables:

- Nuitka (Cross-platform)
- PyInstaller (Cross-platform)
- PyOxidizer (Cross-platform)
- cx_Freeze (Cross-platform)
- py2app (macOS only)
- py2exe (Windows only)

Are there coding standards or a style guide for Python programs?

Yes. The coding style required for standard library modules is documented as **PEP 8**.

Core Language

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

It can be a surprise to get the UnboundLocalError in previously working code when it is modified by adding an assignment statement somewhere in the body of a function.

This code:

```
>>> x = 10
>>> def bar():
...     print(x)
>>> bar()
10
```

works, but this code:

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

results in an UnboundLocalError:

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

This is because 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
```

This explicit declaration is required in order to remind you that (unlike the superficially analogous situation with class and instance variables) you are actually modifying the value of the variable in the outer scope:

```
>>> print(x)
11
```

You can do a similar thing in a nested scope using the `nonlocal` keyword:

```
>>> def foo():
...     x = 10
...     def bar():
...         nonlocal x
...         print(x)
...         x += 1
...     bar()
...     print(x)
>>> foo()
10
11
```

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.

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.

Why do lambdas defined in a loop with different values all return the same result?

Assume you use a for loop to define a few different lambdas (or even plain functions), e.g.:

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

This gives you a list that contains 5 lambdas that calculate `x**2`. You might expect that, when called, they would return, respectively, `0`, `1`, `4`, `9`, and `16`. However, when you actually try you will see that they all return `16`:

```
>>> 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`. You can also verify this by changing the value of `x` and see how the results of the lambdas change:

```
>>> x = 8
>>> squares[2]()
64
```

In order to avoid this, you need to save the values in variables local to the lambdas, so that they don't rely on the value of the global `x`:

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

Here, `n=x` creates a new variable `n` local to the lambda and computed when the lambda is defined so that it has the same value that `x` had at that point in the loop. This means that the value of `n` will be `0` in the first lambda, `1` in the second, `2` in the third, and so on. Therefore each lambda will now return the correct result:

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

How do I share global variables across modules?

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.

What are the “best practices” for using import in a module?

In general, don’t use `from modulename import *`. Doing so clutters the importer’s namespace, and makes it much harder for linters to detect undefined names.

Import modules at the top of a file. Doing so makes it clear what other modules your code requires and avoids questions of whether the module name is in scope. Using one import per line makes it easy to add and delete module imports, but using multiple imports per line uses less screen space.

It’s good practice if you import modules in the following order:

1. standard library modules – e.g. `sys`, `os`, `getopt`, `re`
2. third-party library modules (anything installed in Python’s site-packages directory) – e.g. mx.DateTime, ZODB, PIL.Image, etc.
3. locally-developed modules

It is sometimes necessary to move imports to a function or class to avoid problems with circular imports. Gordon McMillan says:

> Circular imports are fine where both modules use the “import <module>” form of import. They fail when the 2nd module wants to grab a name out of the first (“from module import name”) and the import is at the top level. That’s because names in the 1st are not yet available, because the first module is busy importing the 2nd.

In this case, if the second module is only used in one function, then the import can easily be moved into that function. By the time the import is called, the first module will have finished initializing, and the second module can do its import.

It may also be necessary to move imports out of the top level of code if some of the modules are

the file. In this case, importing the correct modules in the corresponding platform-specific code is a good option.

Only move imports into a local scope, such as inside a function definition, if it's necessary to solve a problem such as avoiding a circular import or are trying to reduce the initialization time of a module. This technique is especially helpful if many of the imports are unnecessary depending on how the program executes. You may also want to move imports into a function if the modules are only ever used in that function. Note that loading a module the first time may be expensive because of the one time initialization of the module, but loading a module multiple times is virtually free, costing only a couple of dictionary lookups. Even if the module name has gone out of scope, the module is probably available in `sys.modules`.

Why are default values shared between objects?

This type of bug commonly bites neophyte programmers. Consider this function:

```
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.

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.

By definition, immutable objects such as numbers, strings, tuples, and `None`, are safe from change. Changes to mutable objects such as dictionaries, lists, and class instances can lead to confusion.

Because of this feature, 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. For example, don't write:

```
def foo(mydict={}):
    ...
```

but:

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

This feature can be useful. When you have a function that's time-consuming to compute, a common technique is to cache the parameters and the resulting value of each call to the function, and return the cached value if the same value is requested again. This is called "memoizing", and can be implemented like this:

```
# Callers can only provide two parameters and optionally pass _cache by keyword
def expensive(arg1, arg2, *, _cache={}):
    if (arg1, arg2) in _cache:
```

```
    # Calculate the value
    result = ... expensive computation ...
    _cache[(arg1, arg2)] = result           # Store result in the cache
    return result
```

You could use a global variable containing a dictionary instead of the default value; it's a matter of taste.

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

Collect the arguments using the `*` and `**` specifiers 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, *args, **kwargs):
    ...
    kwargs['width'] = '14.3c'
    ...
    g(x, *args, **kwargs)
```

What is the difference between arguments and parameters?

Parameters are defined by the names that appear in a function definition, whereas arguments are the values actually passed to a function when calling it. Parameters define what types of arguments a function can accept. For example, given the function definition:

```
def func(foo, bar=None, **kwargs):
    pass
```

foo, *bar* and *kwargs* are parameters of `func`. However, when calling `func`, for example:

```
func(42, bar=314, extra=somevar)
```

the values `42`, `314`, and `somevar` are arguments.

Why did changing list 'y' also change list 'x'?

If you wrote code like:

```
>>> x = []
>>> y = x
>>> y.append(10)
>>> y
[10]
>>> x
[10]
```

you might be wondering why appending an element to `y` changed `x` too.

There are two factors that produce this result:

1. Variables are simply names that refer to objects. Doing `y = x` doesn't create a copy of the list – it creates a new variable `y` that refers to the same object `x` refers to. This means that there is only one object (the list), and both `x` and `y` refer to it.

After the call to `append()`, the content of the mutable object has changed from `[]` to `[10]`. Since both the variables refer to the same object, using either name accesses the modified value `[10]`.

If we instead assign an immutable object to `x`:

```
>>> x = 5  # ints are immutable
>>> y = x
>>> x = x + 1  # 5 can't be mutated, we are creating a new object here
>>> x
6
>>> y
5
```

we can see that in this case `x` and `y` are not equal anymore. This is because integers are immutable, and when we do `x = x + 1` we are not mutating the int `5` by incrementing its value; instead, we are creating a new object (the int `6`) and assigning it to `x` (that is, changing which object `x` refers to). After this assignment we have two objects (the ints `6` and `5`) and two variables that refer to them (`x` now refers to `6` but `y` still refers to `5`).

Some operations (for example `y.append(10)` and `y.sort()`) mutate the object, whereas superficially similar operations (for example `y = y + [10]` and `sorted(y)`) create a new object. In general in Python (and in all cases in the standard library) a method that mutates an object will return `None` to help avoid getting the two types of operations confused. So if you mistakenly write `y.sort()` thinking it will give you a sorted copy of `y`, you'll instead end up with `None`, which will likely cause your program to generate an easily diagnosed error.

However, there is one class of operations where the same operation sometimes has different behaviors with different types: the augmented assignment operators. For example, `+=` mutates lists but not tuples or ints (`a_list += [1, 2, 3]` is equivalent to `a_list.extend([1, 2, 3])` and mutates `a_list`, whereas `some_tuple += (1, 2, 3)` and `some_int += 1` create new objects).

In other words:

- If we have a mutable object (`list`, `dict`, `set`, etc.), we can use some specific operations to mutate it and all the variables that refer to it will see the change.
- If we have an immutable object (`str`, `int`, `tuple`, etc.), all the variables that refer to it will always see the same value, but operations that transform that value into a new value always return a new object.

If you want to know if two variables refer to the same object or not, you can use the `is` operator, or the built-in function `id()`.

How do I write a function with output parameters (call by reference)?

Remember that arguments are passed by assignment in Python. Since assignment just creates references to objects, there's no alias between an argument name in the caller and callee, and so no call-by-reference per se. You can achieve the desired effect in a number of ways.

1. By returning a tuple of the results:

```
>>> def func1(a, b):
...     a = 'new-value'        # a and b are local names
...     b = b + 1              # assigned to new objects
...     return a, b            # return new values
```

```
>>> func1(x, y)
('new-value', 100)
```

This is almost always the clearest solution.

2. By using global variables. This isn't thread-safe, and is not recommended.

3. By passing a mutable (changeable in-place) object:

```
>>> def func2(a):
...     a[0] = 'new-value'     # 'a' references a mutable list
...     a[1] = a[1] + 1        # changes a shared object
...
>>> args = ['old-value', 99]
>>> func2(args)
>>> args
['new-value', 100]
```

4. By passing in a dictionary that gets mutated:

```
>>> def func3(args):
...     args['a'] = 'new-value'     # args is a mutable dictionary
...     args['b'] = args['b'] + 1   # change it in-place
...
>>> args = {'a': 'old-value', 'b': 99}
>>> func3(args)
>>> args
{'a': 'new-value', 'b': 100}
```

5. Or bundle up values in a class instance:

```
>>> class Namespace:
...     def __init__(self, /, **args):
...         for key, value in args.items():
...             setattr(self, key, value)
...
>>> def func4(args):
...     args.a = 'new-value'        # args is a mutable Namespace
...     args.b = args.b + 1         # change object in-place
...
>>> args = Namespace(a='old-value', b=99)
>>> func4(args)
>>> vars(args)
{'a': 'new-value', 'b': 100}
```

There's almost never a good reason to get this complicated.

Your best choice is to return a tuple containing the multiple results.

How do you make a higher order function in Python?

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):
```

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.

How do I copy an object in Python?

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:

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

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.

How can my code discover the name of an object?

Generally speaking, it can't, because objects don't really have names. Essentially, assignment always binds a name to a value; the same is true of `def` and `class` statements, but in that case the value is a callable. Consider the following code:

```
>>> class A:
...     pass
...
>>> B = A
>>> a = B()
>>> b = a
>>> print(b)
<__main__.A object at 0x16D07CC>
>>> print(a)
<__main__.A object at 0x16D07CC>
```

Arguably the class has a name: even though it is bound to two names and invoked through the name B the created instance is still reported as an instance of class A. However, it is impossible to say whether the instance's name is a or b, since both names are bound to the same value.

Generally speaking it should not be necessary for your code to "know the names" of particular values. Unless you are deliberately writing introspective programs, this is usually an indication that a change of approach might be beneficial.

In comp.lang.python, Fredrik Lundh once gave an excellent analogy in answer to this question:

> The same way as you get the name of that cat you found on your porch: the cat (object) itself cannot tell you its name, and it doesn't really care – so the only way to find out what it's called is to ask all your neighbours (namespaces) if it's their cat (object)...
>
> ....and don't be surprised if you'll find that it's known by many names, or no name at all!

What's up with the comma operator's precedence?

Comma is not an operator in Python. Consider this session:

```
>>> "a" in "b", "a"
(False, 'a')
```

Since the comma is not an operator, but a separator between expressions the above is evaluated as if you had entered:

```
("a" in "b"), "a"
```

not:

```
"a" in ("b", "a")
```

The same is true of the various assignment operators (`+=` etc). They are not truly operators but

Is there an equivalent of C's "?:" ternary operator?

Yes, there is. The syntax is as follows:

```
[on_true] if [expression] else [on_false]

x, y = 50, 25
small = x if x < y else y
```

Before this syntax was introduced in Python 2.5, a common idiom was to use logical operators:

```
[expression] and [on_true] or [on_false]
```

However, this idiom is unsafe, as it can give wrong results when *on_true* has a false boolean value. Therefore, it is always better to use the `... if ... else ...` form.

Is it possible to write obfuscated one-liners in Python?

Yes. Usually this is done by nesting `lambda` within `lambda`. See the following three examples, due to Ulf Bartelt:

```
from functools import reduce

# Primes < 1000
print(list(filter(None,map(lambda y:y*reduce(lambda x,y:x*y!=0,
map(lambda x,y=y:y%x,range(2,int(pow(y,0.5)+1))),1),range(2,1000)))))

# First 10 Fibonacci numbers
print(list(map(lambda x,f=lambda x,f:(f(x-1,f)+f(x-2,f)) if x>1 else 1:
f(x,f), range(10))))

# Mandelbrot set
print((lambda Ru,Ro,Iu,Io,IM,Sx,Sy:reduce(lambda x,y:x+y,map(lambda y,
Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,Sy=Sy,L=lambda yc,Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,i=IM,
Sx=Sx,Sy=Sy:reduce(lambda x,y:x+y,map(lambda x,xc=Ru,yc=yc,Ru=Ru,Ro=Ro,
i=i,Sx=Sx,F=lambda xc,yc,x,y,k,f=lambda xc,yc,x,y,k,f:(k<=0)or (x*x+y*y
>=4.0) or 1+f(xc,yc,x*x-y*y+xc,2.0*x*y+yc,k-1,f):f(xc,yc,x,y,k,f):chr(
64+F(Ru+x*(Ro-Ru)/Sx,yc,0,0,i)),range(Sx))):L(Iu+y*(Io-Iu)/Sy),range(Sy
))))(-2.1, 0.7, -1.2, 1.2, 30, 80, 24))
#     \___ ___/  \___ ___/  |   |   |__ lines on screen
#         V          V      |   |______ columns on screen
#         |          |      |_________ maximum of "iterations"
#         |          |________________ range on y axis
#         |___________________________ range on x axis
```

Don't try this at home, kids!

What does the slash(/) in the parameter list of a function mean?

A slash in the argument list of a function denotes that the parameters prior to it are positional-only. Positional-only parameters are the ones without an externally-usable name. Upon calling a function that accepts positional-only parameters, arguments are mapped to parameters based solely on their position. For example, `divmod()` is a function that accepts positional-only parameters. Its documentation looks like this:

```
>>>
```

```
divmod(x, y, /)
    Return the tuple (x//y, x%y).  Invariant: div*y + mod == x.
```

The slash at the end of the parameter list means that both parameters are positional-only. Thus, calling `divmod()` with keyword arguments would lead to an error:

```
>>> divmod(x=3, y=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: divmod() takes no keyword arguments
```

Numbers and strings

How do I specify hexadecimal and octal integers?

To specify an octal digit, precede the octal value with a zero, and then a lower or uppercase “o”. For example, to set the variable “a” to the octal value “10” (8 in decimal), type:

```
>>> a = 0o10
>>> a
8
```

Hexadecimal is just as easy. Simply precede the hexadecimal number with a zero, and then a lower or uppercase “x”. Hexadecimal digits can be specified in lower or uppercase. For example, in the Python interpreter:

```
>>> a = 0xa5
>>> a
165
>>> b = 0XB2
>>> b
178
```

Why does -22 // 10 return -3?

It’s primarily driven by the desire that `i % j` have the same sign as `j`. If you want that, and also want:

```
i == (i // j) * j + (i % j)
```

then integer division has to return the floor. C also requires that identity to hold, and then compilers that truncate `i // j` need to make `i % j` have the same sign as `i`.

There are few real use cases for `i % j` when `j` is negative. When `j` is positive, there are many, and in virtually all of them it’s more useful for `i % j` to be `>= 0`. If the clock says 10 now, what did it say 200 hours ago? `-190 % 12 == 2` is useful; `-190 % 12 == -10` is a bug waiting to bite.

How do I get int literal attribute instead of SyntaxError?

Trying to lookup an `int` literal attribute in the normal manner gives a syntax error because the period is seen as a decimal point:

```
>>> 1.__class__
```

```
    ^
SyntaxError: invalid decimal literal
```

The solution is to separate the literal from the period with either a space or parentheses.

```
>>> 1 .__class__
<class 'int'>
>>> (1).__class__
<class 'int'>
```

How do I convert a string to a number?

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` holds true, 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 '0o' 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 does not allow leading '0' in a decimal number (except '0').

How do I convert a number to a string?

To convert, e.g., the number 144 to the string '144', use the built-in type constructor `str()`. If you want a hexadecimal or octal representation, use the built-in functions `hex()` or `oct()`. For fancy formatting, see the Formatted string literals and Format String Syntax sections, e.g. `"{:04d}".format(144)` yields `'0144'` and `"{:.3f}".format(1.0/3.0)` yields `'0.333'`.

How do I modify a string in place?

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. However, if you need an object with the ability to modify in-place unicode data, try using an `io.StringIO` object or the `array` module:

```
>>> import io
>>> s = "Hello, world"
>>> sio = io.StringIO(s)
>>> sio.getvalue()
'Hello, world'
>>> sio.seek(7)
7
>>> sio.write("there!")
6
>>> sio.getvalue()
'Hello, there!'

>>> import array
```

```
array('u', 'Hello, world')
>>> a[0] = 'y'
>>> print(a)
array('u', 'yello, world')
>>> a.tounicode()
'yello, world'
```

How do I use strings to call functions/methods?

There are various techniques.

- The best is to use a dictionary that maps strings to functions. The primary advantage of this technique is that the strings do not need to match the names of the functions. This is also the primary technique used to emulate a case construct:

```
def a():
    pass

def b():
    pass

dispatch = {'go': a, 'stop': b}  # Note lack of parens for funcs

dispatch[get_input()]()  # Note trailing parens to call function
```

- Use the built-in function `getattr()`:

```
import foo
getattr(foo, 'bar')()
```

  Note that `getattr()` works on any object, including classes, class instances, modules, and so on.

  This is used in several places in the standard library, like this:

```
class Foo:
    def do_foo(self):
        ...

    def do_bar(self):
        ...

f = getattr(foo_instance, 'do_' + opname)
f()
```

- Use `locals()` to resolve the function name:

```
def myFunc():
    print("hello")

fname = "myFunc"

f = locals()[fname]
f()
```

Is there an equivalent to Perl's chomp() for removing trailing newlines from strings?

You can use `S.rstrip("\r\n")` to remove all occurrences of any line terminator from the end of the string `S` without removing other trailing whitespace. If the string `S` represents more than one line, with several empty lines at the end, the line terminators for all the blank lines will be removed:

```
>>> lines = ("line 1 \r\n"
...          "\r\n"
...          "\r\n")
>>> lines.rstrip("\n\r")
'line 1 '
```

Since this is typically only desired when reading text one line at a time, using `S.rstrip()` this way works well.

Is there a scanf() or sscanf() equivalent?

Not as such.

For simple input parsing, the easiest approach is usually to split the line into whitespace-delimited words using the `split()` method of string objects and then convert decimal strings to numeric values using `int()` or `float()`. `split()` supports an optional “sep” parameter which is useful if the line uses something other than whitespace as a separator.

For more complicated input parsing, regular expressions are more powerful than C’s `sscanf()` and better suited for the task.

What does ‘UnicodeDecodeError’ or ‘UnicodeEncodeError’ error mean?

See the Unicode HOWTO.

Performance

My program is too slow. How do I speed it up?

That’s a tough one, in general. First, here are a list of things to remember before diving further:

- Performance characteristics vary across Python implementations. This FAQ focuses on CPython.
- Behaviour can vary across operating systems, especially when talking about I/O or multi-threading.
- You should always find the hot spots in your program *before* attempting to optimize any code (see the `profile` module).
- Writing benchmark scripts will allow you to iterate quickly when searching for improvements (see the `timeit` module).
- It is highly recommended to have good code coverage (through unit testing or any other technique) before potentially introducing regressions hidden in sophisticated optimizations.

That being said, there are many tricks to speed up Python code. Here are some general principles which go a long way towards reaching acceptable performance levels:

- Making your algorithms faster (or changing to faster ones) can yield much larger benefits than trying to sprinkle micro-optimization tricks all over your code.
- Use the right data structures. Study documentation for the Built-in Types and the `collections` module.
- When the standard library provides a primitive for doing something, it is likely (although not guaranteed) to be faster than any alternative you may come up with. This is doubly true for

either the `list.sort()` built-in method or the related `sorted()` function to do sorting (and see the Sorting HOW TO for examples of moderately advanced usage).
- Abstractions tend to create indirections and force the interpreter to work more. If the levels of indirection outweigh the amount of useful work done, your program will be slower. You should avoid excessive abstraction, especially under the form of tiny functions or methods (which are also often detrimental to readability).

If you have reached the limit of what pure Python can allow, there are tools to take you further away. For example, Cython can compile a slightly modified version of Python code into a C extension, and can be used on many different platforms. Cython can take advantage of compilation (and optional type annotations) to make your code significantly faster than when interpreted. If you are confident in your C programming skills, you can also write a C extension module yourself.

See also: The wiki page devoted to performance tips.

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

`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, the recommended idiom is 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)
```

(another reasonably efficient idiom is to use `io.StringIO`)

To accumulate many `bytes` objects, the recommended idiom is to extend a `bytearray` object using in-place concatenation (the += operator):

```
result = bytearray()
for b in my_bytes_objects:
    result += b
```

Sequences (Tuples/Lists)

How do I convert between tuples and lists?

The type constructor `tuple(seq)` converts any sequence (actually, any iterable) into a tuple with the same items in the same order.

For example, `tuple([1, 2, 3])` yields `(1, 2, 3)` and `tuple('abc')` yields `('a', 'b', 'c')`. If the argument is a tuple, it does not make a copy but returns the same object, so it is cheap to call `tuple()` when you aren't sure that an object is already a tuple.

The type constructor `list(seq)` converts any sequence or iterable into a list with the same items in the same order. For example, `list((1, 2, 3))` yields `[1, 2, 3]` and `list('abc')` yields `['a', 'b', 'c']`. If the argument is a list, it makes a copy just like `seq[:]` would.

Python sequences are indexed with positive numbers and negative numbers. For positive numbers 0 is the first index 1 is the second index and so forth. For negative indices -1 is the last index and -2 is the penultimate (next to last) index and so forth. Think of `seq[-n]` as the same as `seq[len(seq)-n]`.

Using negative indices can be very convenient. For example `S[:-1]` is all of the string except for its last character, which is useful for removing the trailing newline from a string.

How do I iterate over a sequence in reverse order?

Use the `reversed()` built-in function:

```
for x in reversed(sequence):
    ...  # do something with x ...
```

This won't touch your original sequence, but build a new copy with reversed order to iterate over.

How do you remove duplicates from a list?

See the Python Cookbook for a long discussion of many ways to do this:

https://code.activestate.com/recipes/52560/

If you don't mind reordering the list, sort it and then scan from the end of the list, deleting duplicates as you go:

```
if mylist:
    mylist.sort()
    last = mylist[-1]
    for i in range(len(mylist)-2, -1, -1):
        if last == mylist[i]:
            del mylist[i]
        else:
            last = mylist[i]
```

If all elements of the list may be used as set keys (i.e. they are all hashable) this is often faster

```
mylist = list(set(mylist))
```

This converts the list into a set, thereby removing duplicates, and then back into a list.

How do you remove multiple items from a list

As with removing duplicates, explicitly iterating in reverse with a delete condition is one possibility. However, it is easier and faster to use slice replacement with an implicit or explicit forward iteration. Here are three variations.:

```
mylist[:] = filter(keep_function, mylist)
mylist[:] = (x for x in mylist if keep_condition)
mylist[:] = [x for x in mylist if keep_condition]
```

The list comprehension may be fastest.

How do you make an array in Python?

```
["this", 1, "is", "an", "array"]
```

Lists are equivalent to C or Pascal arrays in their time complexity; the primary difference is that a Python list can contain objects of many different types.

The `array` module also provides methods for creating arrays of fixed types with compact representations, but they are slower to index than lists. Also note that NumPy and other third party packages define array-like structures with various characteristics as well.

To get Lisp-style linked lists, you can emulate cons cells using tuples:

```
lisp_list = ("like",  ("this",  ("example", None) ) )
```

If mutability is desired, you could use lists instead of tuples. Here the analogue of lisp car is `lisp_list[0]` and the analogue of cdr is `lisp_list[1]`. Only do this if you're sure you really need to, because it's usually a lot slower than using Python lists.

How do I create a multidimensional list?

You probably tried to make a multidimensional array like this:

```
>>> A = [[None] * 2] * 3
```

This looks correct if you print it:

```
>>> A
[[None, None], [None, None], [None, None]]
```

But when you assign a value, it shows up in multiple places:

```
>>> A[0][0] = 5
>>> A
[[5, None], [5, None], [5, None]]
```

The reason is that replicating a list with `*` doesn't create copies, it only creates references to the existing objects. The `*3` creates a list containing 3 references to the same list of length two. Changes to one row will show in all rows, which is almost certainly not what you want.

The suggested approach is to create a list of the desired length first and then fill in each element with a newly created list:

```
A = [None] * 3
for i in range(3):
    A[i] = [None] * 2
```

This generates a list containing 3 different lists of length two. You can also use a list comprehension:

```
w, h = 2, 3
A = [[None] * w for i in range(h)]
```

Or, you can use an extension that provides a matrix datatype; NumPy is the best known.

How do I apply a method to a sequence of objects?

Use a list comprehension:

```
result = [obj.method() for obj in mylist]
```

Why does a_tuple[i] += [‘item’] raise an exception when the addition works?

This is because of a combination of the fact that augmented assignment operators are *assignment* operators, and the difference between mutable and immutable objects in Python.

This discussion applies in general when augmented assignment operators are applied to elements of a tuple that point to mutable objects, but we’ll use a `list` and `+=` as our exemplar.

If you wrote:

```
>>> a_tuple = (1, 2)
>>> a_tuple[0] += 1
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

The reason for the exception should be immediately clear: `1` is added to the object `a_tuple[0]` points to (`1`), producing the result object, `2`, but when we attempt to assign the result of the computation, `2`, to element `0` of the tuple, we get an error because we can’t change what an element of a tuple points to.

Under the covers, what this augmented assignment statement is doing is approximately this:

```
>>> result = a_tuple[0] + 1
>>> a_tuple[0] = result
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

It is the assignment part of the operation that produces the error, since a tuple is immutable.

When you write something like:

```
>>> a_tuple = (['foo'], 'bar')
>>> a_tuple[0] += ['item']
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

The exception is a bit more surprising, and even more surprising is the fact that even though there was an error, the append worked:

```
>>> a_tuple[0]
['foo', 'item']
```

To see why this happens, you need to know that (a) if an object implements an `__iadd__` magic method, it gets called when the `+=` augmented assignment is executed, and its return value is what gets used in the assignment statement; and (b) for lists, `__iadd__` is equivalent to calling `extend` on the list and returning the list. That’s why we say that for lists, `+=` is a “shorthand” for `list.extend`:

```
>>> a_list
[1]
```

This is equivalent to:

```
>>> result = a_list.__iadd__([1])
>>> a_list = result
```

The object pointed to by a_list has been mutated, and the pointer to the mutated object is assigned back to `a_list`. The end result of the assignment is a no-op, since it is a pointer to the same object that `a_list` was previously pointing to, but the assignment still happens.

Thus, in our tuple example what is happening is equivalent to:

```
>>> result = a_tuple[0].__iadd__(['item'])
>>> a_tuple[0] = result
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

The `__iadd__` succeeds, and thus the list is extended, but even though `result` points to the same object that `a_tuple[0]` already points to, that final assignment still results in an error, because tuples are immutable.

I want to do a complicated sort: can you do a Schwartzian Transform in Python?

The technique, attributed to Randal Schwartz of the Perl community, sorts the elements of a list by a metric which maps each element to its "sort value". In Python, use the `key` argument for the `list.sort()` method:

```
Isorted = L[:]
Isorted.sort(key=lambda s: int(s[10:15]))
```

How can I sort one list by values from another list?

Merge them into an iterator of tuples, sort the resulting list, and then pick out the element you want.

```
>>> list1 = ["what", "I'm", "sorting", "by"]
>>> list2 = ["something", "else", "to", "sort"]
>>> pairs = zip(list1, list2)
>>> pairs = sorted(pairs)
>>> pairs
[("I'm", 'else'), ('by', 'sort'), ('sorting', 'to'), ('what', 'something')]
>>> result = [x[1] for x in pairs]
>>> result
['else', 'sort', 'to', 'something']
```

Objects

What is a class?

A class is the particular object type created by executing a class statement. Class objects are used as templates to create instance objects, which embody both the data (attributes) and code (methods) specific to a datatype.

A class can be based on one or more other classes, called its base class(es). It then inherits the attributes and methods of its base classes. This allows an object model to be successively refined by inheritance. You might have a generic `Mailbox` class that provides basic accessor methods for a mailbox, and subclasses such as `MboxMailbox`, `MaildirMailbox`, `OutlookMailbox` that handle various specific mailbox formats.

What is a method?

A method is a function on some object `x` that you normally call as `x.name(arguments...)`. Methods are defined as functions inside the class definition:

```
class C:
    def meth(self, arg):
        return arg * 2 + self.attribute
```

What is self?

Self is merely a conventional name for the first argument of a method. A method defined as `meth(self, a, b, c)` should be called as `x.meth(a, b, c)` for some instance `x` of the class in which the definition occurs; the called method will think it is called as `meth(x, a, b, c)`.

See also Why must 'self' be used explicitly in method definitions and calls?.

How do I check if an object is an instance of a given class or of a subclass of it?

Use the built-in function `isinstance(obj, cls)`. You can check if an object is an instance of any of a number of classes by providing a tuple instead of a single class, e.g. `isinstance(obj, (class1, class2, ...))`, and can also check whether an object is one of Python's built-in types, e.g. `isinstance(obj, str)` or `isinstance(obj, (int, float, complex))`.

Note that `isinstance()` also checks for virtual inheritance from an abstract base class. So, the test will return `True` for a registered class even if hasn't directly or indirectly inherited from it. To test for "true inheritance", scan the MRO of the class:

```
from collections.abc import Mapping

class P:
     pass

class C(P):
    pass

Mapping.register(P)
```

```
>>> c = C()
>>> isinstance(c, C)        # direct
True
>>> isinstance(c, P)        # indirect
True
>>> isinstance(c, Mapping)  # virtual
True

# Actual inheritance chain
>>> type(c).__mro__
```

```
# Test for "true inheritance"
>>> Mapping in type(c).__mro__
False
```

Note that most programs do not use `isinstance()` on user-defined classes very often. If you are developing the classes yourself, a more proper object-oriented style is to define methods on the classes that encapsulate a particular behaviour, instead of checking the object's class and doing a different thing based on what class it is. For example, if you have a function that does something:

```
def search(obj):
    if isinstance(obj, Mailbox):
        ...  # code to search a mailbox
    elif isinstance(obj, Document):
        ...  # code to search a document
    elif ...
```

A better approach is to define a `search()` method on all the classes and just call it:

```
class Mailbox:
    def search(self):
        ...  # code to search a mailbox

class Document:
    def search(self):
        ...  # code to search a document

obj.search()
```

What is delegation?

Delegation is an object oriented technique (also called a design pattern). Let's say you have an object `x` and want to change the behaviour of just one of its methods. You can create a new class that provides a new implementation of the method you're interested in changing and delegates all other methods to the corresponding method of `x`.

Python programmers can easily implement delegation. For example, the following class implements a class that behaves like a file but converts all written data to uppercase:

```
class UpperOut:

    def __init__(self, outfile):
        self._outfile = outfile

    def write(self, s):
        self._outfile.write(s.upper())

    def __getattr__(self, name):
        return getattr(self._outfile, name)
```

Here the `UpperOut` class redefines the `write()` method to convert the argument string to uppercase before calling the underlying `self._outfile.write()` method. All other methods are delegated to the underlying `self._outfile` object. The delegation is accomplished via the `__getattr__` method; consult the language reference for more information about controlling attribute access.

Note that for more general cases delegation can get trickier. When attributes must be set as well as

```
class X:
    ...
    def __setattr__(self, name, value):
        self.__dict__[name] = value
    ...
```

Most `__setattr__()` implementations must modify `self.__dict__` to store local state for self without causing an infinite recursion.

How do I call a method defined in a base class from a derived class that extends it?

Use the built-in `super()` function:

```
class Derived(Base):
    def meth(self):
        super().meth()  # calls Base.meth
```

In the example, `super()` will automatically determine the instance from which it was called (the `self` value), look up the method resolution order (MRO) with `type(self).__mro__`, and return the next in line after `Derived` in the MRO: `Base`.

How can I organize my code to make it easier to change the base class?

You could assign the base class to an alias and derive from the alias. Then all you have to change is the value assigned to the alias. Incidentally, this trick is also handy if you want to decide dynamically (e.g. depending on availability of resources) which base class to use. Example:

```
class Base:
    ...

BaseAlias = Base

class Derived(BaseAlias):
    ...
```

How do I create static class data and static class methods?

Both static data and static methods (in the sense of C++ or Java) are supported in Python.

For static data, simply define a class attribute. To assign a new value to the attribute, you have to explicitly use the class name in the assignment:

```
class C:
    count = 0   # number of times C.__init__ called

    def __init__(self):
        C.count = C.count + 1

    def getcount(self):
        return C.count  # or return self.count
```

`c.count` also refers to `C.count` for any `c` such that `isinstance(c, C)` holds, unless overridden by `c` itself or by some class on the base-class search path from `c.__class__` back to `C`.

Caution: within a method of C, an assignment like `self.count = 42` creates a new and unrelated

specify the class whether inside a method or not:

```
C.count = 314
```

Static methods are possible:

```
class C:
    @staticmethod
    def static(arg1, arg2, arg3):
        # No 'self' parameter!
        ...
```

However, a far more straightforward way to get the effect of a static method is via a simple module-level function:

```
def getcount():
    return C.count
```

If your code is structured so as to define one class (or tightly related class hierarchy) per module, this supplies the desired encapsulation.

How can I overload constructors (or methods) in Python?

This answer actually applies to all methods, but the question usually comes up first in the context of constructors.

In C++ you'd write

```
class C {
    C() { cout << "No arguments\n"; }
    C(int i) { cout << "Argument is " << i << "\n"; }
}
```

In Python you have to write a single constructor that catches all cases using default arguments. For example:

```
class C:
    def __init__(self, i=None):
        if i is None:
            print("No arguments")
        else:
            print("Argument is", i)
```

This is not entirely equivalent, but close enough in practice.

You could also try a variable-length argument list, e.g.

```
def __init__(self, *args):
    ...
```

The same approach works for all method definitions.

I try to use __spam and I get an error about _SomeClassName__spam.

Variable names with double leading underscores are "mangled" to provide a simple but effective way

at most one trailing underscore) is textually replaced with `_classname__spam`, where `classname` is the current class name with any leading underscores stripped.

This doesn't guarantee privacy: an outside user can still deliberately access the "_classname__spam" attribute, and private values are visible in the object's `__dict__`. Many Python programmers never bother to use private variable names at all.

My class defines __del__ but it is not called when I delete the object.

There are several possible reasons for this.

The del statement does not necessarily call `__del__()` – it simply decrements the object's reference count, and if this reaches zero `__del__()` is called.

If your data structures contain circular links (e.g. a tree where each child has a parent reference and each parent has a list of children) the reference counts will never go back to zero. Once in a while Python runs an algorithm to detect such cycles, but the garbage collector might run some time after the last reference to your data structure vanishes, so your `__del__()` method may be called at an inconvenient and random time. This is inconvenient if you're trying to reproduce a problem. Worse, the order in which object's `__del__()` methods are executed is arbitrary. You can run `gc.collect()` to force a collection, but there *are* pathological cases where objects will never be collected.

Despite the cycle collector, it's still a good idea to define an explicit `close()` method on objects to be called whenever you're done with them. The `close()` method can then remove attributes that refer to subobjects. Don't call `__del__()` directly – `__del__()` should call `close()` and `close()` should make sure that it can be called more than once for the same object.

Another way to avoid cyclical references is to use the `weakref` module, which allows you to point to objects without incrementing their reference count. Tree data structures, for instance, should use weak references for their parent and sibling references (if they need them!).

Finally, if your `__del__()` method raises an exception, a warning message is printed to `sys.stderr`.

How do I get a list of all instances of a given class?

Python does not keep track of all instances of a class (or of a built-in type). You can program the class's constructor to keep track of all instances by keeping a list of weak references to each instance.

Why does the result of `id()` appear to be not unique?

The `id()` builtin returns an integer that is guaranteed to be unique during the lifetime of the object. Since in CPython, this is the object's memory address, it happens frequently that after an object is deleted from memory, the next freshly created object is allocated at the same position in memory. This is illustrated by this example:

```
>>> id(1000)
13901272
>>> id(2000)
13901272
```

The two ids belong to different integer objects that are created before, and deleted immediately

```
>>> a = 1000; b = 2000
>>> id(a)
13901272
>>> id(b)
13891296
```

When can I rely on identity tests with the *is* operator?

The `is` operator tests for object identity. The test `a is b` is equivalent to `id(a) == id(b)`.

The most important property of an identity test is that an object is always identical to itself, `a is a` always returns `True`. Identity tests are usually faster than equality tests. And unlike equality tests, identity tests are guaranteed to return a boolean `True` or `False`.

However, identity tests can *only* be substituted for equality tests when object identity is assured. Generally, there are three circumstances where identity is guaranteed:

1) Assignments create new names but do not change object identity. After the assignment `new = old`, it is guaranteed that `new is old`.

2) Putting an object in a container that stores object references does not change object identity. After the list assignment `s[0] = x`, it is guaranteed that `s[0] is x`.

3) If an object is a singleton, it means that only one instance of that object can exist. After the assignments `a = None` and `b = None`, it is guaranteed that `a is b` because `None` is a singleton.

In most other circumstances, identity tests are inadvisable and equality tests are preferred. In particular, identity tests should not be used to check constants such as `int` and `str` which aren't guaranteed to be singletons:

```
>>> a = 1000
>>> b = 500
>>> c = b + 500
>>> a is c
False

>>> a = 'Python'
>>> b = 'Py'
>>> c = b + 'thon'
>>> a is c
False
```

Likewise, new instances of mutable containers are never identical:

```
>>> a = []
>>> b = []
>>> a is b
False
```

In the standard library code, you will see several common patterns for correctly using identity tests:

1) As recommended by **PEP 8**, an identity test is the preferred way to check for `None`. This reads like plain English in code and avoids confusion with other objects that may have boolean values that evaluate to false.

here is how to implement a method that behaves like `dict.pop()`:

```
_sentinel = object()

def pop(self, key, default=_sentinel):
    if key in self:
        value = self[key]
        del self[key]
        return value
    if default is _sentinel:
        raise KeyError(key)
    return default
```

3) Container implementations sometimes need to augment equality tests with identity tests. This prevents the code from being confused by objects such as `float('NaN')` that are not equal to themselves.

For example, here is the implementation of `collections.abc.Sequence.__contains__()`:

```
def __contains__(self, value):
    for v in self:
        if v is value or v == value:
            return True
    return False
```

How can a subclass control what data is stored in an immutable instance?

When subclassing an immutable type, override the `__new__()` method instead of the `__init__()` method. The latter only runs *after* an instance is created, which is too late to alter data in an immutable instance.

All of these immutable classes have a different signature than their parent class:

```
from datetime import date

class FirstOfMonthDate(date):
    "Always choose the first day of the month"
    def __new__(cls, year, month, day):
        return super().__new__(cls, year, month, 1)

class NamedInt(int):
    "Allow text names for some numbers"
    xlat = {'zero': 0, 'one': 1, 'ten': 10}
    def __new__(cls, value):
        value = cls.xlat.get(value, value)
        return super().__new__(cls, value)

class TitleStr(str):
    "Convert str to name suitable for a URL path"
    def __new__(cls, s):
        s = s.lower().replace(' ', '-')
        s = ''.join([c for c in s if c.isalnum() or c == '-'])
        return super().__new__(cls, s)
```

The classes can be used like this:

```
>>> FirstOfMonthDate(2012, 2, 14)
```

```
>>> NamedInt('ten')
10
>>> NamedInt(20)
20
>>> TitleStr('Blog: Why Python Rocks')
'blog-why-python-rocks'
```

How do I cache method calls?

The two principal tools for caching methods are `functools.cached_property()` and `functools.lru_cache()`. The former stores results at the instance level and the latter at the class level.

The *cached_property* approach only works with methods that do not take any arguments. It does not create a reference to the instance. The cached method result will be kept only as long as the instance is alive.

The advantage is that when an instance is no longer used, the cached method result will be released right away. The disadvantage is that if instances accumulate, so too will the accumulated method results. They can grow without bound.

The *lru_cache* approach works with methods that have hashable arguments. It creates a reference to the instance unless special efforts are made to pass in weak references.

The advantage of the least recently used algorithm is that the cache is bounded by the specified *maxsize*. The disadvantage is that instances are kept alive until they age out of the cache or until the cache is cleared.

This example shows the various techniques:

```
class Weather:
    "Lookup weather information on a government website"

    def __init__(self, station_id):
        self._station_id = station_id
        # The _station_id is private and immutable

    def current_temperature(self):
        "Latest hourly observation"
        # Do not cache this because old results
        # can be out of date.

    @cached_property
    def location(self):
        "Return the longitude/latitude coordinates of the station"
        # Result only depends on the station_id

    @lru_cache(maxsize=20)
    def historic_rainfall(self, date, units='mm'):
        "Rainfall on a given date"
        # Depends on the station_id, date, and units.
```

The above example assumes that the *station_id* never changes. If the relevant instance attributes are mutable, the *cached_property* approach can't be made to work because it cannot detect changes to the attributes.

The *lru_cache* approach can be made to work, but the class needs to define the *__eq__* and *__hash__*

```
class Weather:
    "Example with a mutable station identifier"

    def __init__(self, station_id):
        self.station_id = station_id

    def change_station(self, station_id):
        self.station_id = station_id

    def __eq__(self, other):
        return self.station_id == other.station_id

    def __hash__(self):
        return hash(self.station_id)

    @lru_cache(maxsize=20)
    def historic_rainfall(self, date, units='cm'):
        'Rainfall on a given date'
        # Depends on the station_id, date, and units.
```

Modules

How do I create a .pyc file?

When a module is imported for the first time (or when the source file has changed since the current compiled file was created) a `.pyc` file containing the compiled code should be created in a `__pycache__` subdirectory of the directory containing the `.py` file. The `.pyc` file will have a filename that starts with the same name as the `.py` file, and ends with `.pyc`, with a middle component that depends on the particular `python` binary that created it. (See **PEP 3147** for details.)

One reason that a `.pyc` file may not be created is a permissions problem with the directory containing the source file, meaning that the `__pycache__` subdirectory cannot be created. This can happen, for example, if you develop as one user but run as another, such as if you are testing with a web server.

Unless the `PYTHONDONTWRITEBYTECODE` environment variable is set, creation of a .pyc file is automatic if you're importing a module and Python has the ability (permissions, free space, etc...) to create a `__pycache__` subdirectory and write the compiled module to that subdirectory.

Running Python on a top level script is not considered an import and no `.pyc` will be created. For example, if you have a top-level module `foo.py` that imports another module `xyz.py`, when you run `foo` (by typing `python foo.py` as a shell command), a `.pyc` will be created for `xyz` because `xyz` is imported, but no `.pyc` file will be created for `foo` since `foo.py` isn't being imported.

If you need to create a `.pyc` file for `foo` – that is, to create a `.pyc` file for a module that is not imported – you can, using the `py_compile` and `compileall` modules.

The `py_compile` module can manually compile any module. One way is to use the `compile()` function in that module interactively:

```
>>> import py_compile
>>> py_compile.compile('foo.py')
```

This will write the `.pyc` to a `__pycache__` subdirectory in the same location as `foo.py` (or you can override that with the optional parameter `cfile`).

You can also automatically compile all files in a directory or directories using the `compileall` module. You can do it from the shell prompt by running `compileall.py` and providing the path of a directory containing Python files to compile:

```
python -m compileall .
```

How do I find the current module name?

A module can find out its own module name by looking at the predefined global variable `__name__`. If this has the value `'__main__'`, the program is running as a script. Many modules that are usually used by importing them also provide a command-line interface or a self-test, and only execute this code after checking `__name__`:

```
def main():
    print('Running test...')
    ...

if __name__ == '__main__':
    main()
```

How can I have modules that mutually import each other?

Suppose you have the following modules:

`foo.py`:

```
from bar import bar_var
foo_var = 1
```

`bar.py`:

```
from foo import foo_var
bar_var = 2
```

The problem is that the interpreter will perform the following steps:

- main imports `foo`
- Empty globals for `foo` are created
- `foo` is compiled and starts executing
- `foo` imports `bar`
- Empty globals for `bar` are created
- `bar` is compiled and starts executing
- `bar` imports `foo` (which is a no-op since there already is a module named `foo`)
- The import mechanism tries to read `foo_var` from `foo` globals, to set `bar.foo_var = foo.foo_var`

The last step fails, because Python isn't done with interpreting `foo` yet and the global symbol dictionary for `foo` is still empty.

The same thing happens when you use `import foo`, and then try to access `foo.foo_var` in global code.

There are (at least) three possible workarounds for this problem.

Guido van Rossum recommends avoiding all uses of `from <module> import ...`, and placing all code inside functions. Initializations of global variables and class variables should use constants or built-in functions only. This means everything from an imported module is referenced as `<module>.<name>`.

Jim Roskind suggests performing steps in the following order in each module:

- exports (globals, functions, and classes that don't need imported base classes)
- `import` statements
- active code (including globals that are initialized from imported values).

Van Rossum doesn't like this approach much because the imports appear in a strange place, but it does work.

Matthias Urlichs recommends restructuring your code so that the recursive import is not necessary in the first place.

These solutions are not mutually exclusive.

__import__('x.y.z') returns <module 'x'>; how do I get z?

Consider using the convenience function `import_module()` from `importlib` instead:

```
z = importlib.import_module('x.y.z')
```

When I edit an imported module and reimport it, the changes don't show up. Why does this happen?

For reasons of efficiency as well as consistency, Python only reads the module file on the first time a module is imported. If it didn't, in a program consisting of many modules where each one imports the same basic module, the basic module would be parsed and re-parsed many times. To force re-reading of a changed module, do this:

```
import importlib
import modname
importlib.reload(modname)
```

Warning: this technique is not 100% fool-proof. In particular, modules containing statements like

```
from modname import some_objects
```

will continue to work with the old version of the imported objects. If the module contains class definitions, existing class instances will *not* be updated to use the new class definition. This can result in the following paradoxical behaviour:

```
>>> import importlib
>>> import cls
>>> c = cls.C()                # Create an instance of C
>>> importlib.reload(cls)
<module 'cls' from 'cls.py'>
>>> isinstance(c, cls.C)       # isinstance is false?!?
False
```

The nature of the problem is made clear if you print out the "identity" of the class objects:

```
>>> hex(id(c.__class__))
'0x7352a0'
>>> hex(id(cls.C))
'0x4198d0'
```

"