Python 3 Programming

Fundamental Variables

FUNDAMENTAL VARIABLES

Contents

Summary

This chapter discusses the basic building blocks of a Python program - its object types and variables.

Python is object oriented

So what is an object?

- To a mathematician the term describes a "thing"
- To a programmer the term describes a specific area of memory

Objects have type, state, and identity

- An object's type is called its *Class* *
  - → Describes the size and format of the area of memory
  - → Describes the actions which may be carried out on the object
- An object's state is the value of data held in the memory
- An object's identity is its unique position (*address*) in memory
  - → Python uses the memory location to identify the object
  - → Python does not expose memory addresses directly

If that is an object, what is a variable?

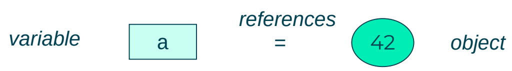
Object orientation is inescapable if you wish to understand Python. You do not need to write OO code, but it is important to understand the principles, which are actually not that complicated. Many programming languages have the concept of variable types, which is roughly analogous to a class - it describes the object to which the variable is referring. How that object is laid out in memory is not particularly important to us, but it is important to Python - it has to know the size and format of memory required, and it is up to us to describe it. Fortunately, the common object types (classes), like strings, files, and exceptions, are already defined.

In Python, an object's identity can be obtained using the id built-in function, although this is rarely needed.

* Are classes types? Not really, we shall discuss this later (hint: duck-typing).

Python variables

- **Python variables are references to objects**



- **Variables are defined automatically**
  → An undefined variable refers to a special object called None
- **Variables can be deleted with `del`**
  → An object's memory can be reused when it is no longer referenced
- **Variables are local by default**
  → If created within a user written function
  → More on `global` variables and scope later...
  → Display local variables with `print(locals())`

Like most scripting languages, Python variables are defined automatically, and are untyped until assigned. Variables are actually references to objects, so the assignment of a string to a variable makes that variable reference a string object. Uninitialised variables reference an object called None (NULL or undef in other languages).
Being objects, class specific functions, like altering the case of a string, are implemented as methods calls on the object. If you are not familiar with this terminology, a method call is like a conventional function call, with a reference to the variable passed automatically. We shall see examples shortly.
Where the objects themselves are immutable (cannot be altered), then several variables may reference the same physical value.
Unlike most scripting languages, variables defined in a function are local by default - and must be specifically marked as global if required.
When you delete a variable then that removes the name. That will decrement the object's reference count, and when the count reaches zero then the memory can be reused.

Variable names

- **Case sensitive**
- **Usual rules for the name**

→ Must start with an underscore or a letter

→ Followed by any number of letters, digits, or underscores

- **Conventions with underscores**

→ Names beginning with one underscore are private to a module/class

```
_private_to_module
```

→ Names beginning with two underscores are mangled

```
__private_to_class
```

→ Names beginning and ending with two underscores are special

```
__itsakindamagic__
```

→ The character _ represents the result of the previous command

The names given to variables, and also to other symbols like functions, follow the usual rules common to many programming languages. For example, names are case sensitive. Variable names must not clash with Python keywords: these may be listed with help ('keywords'), or consult the list at the end of this chapter.

In addition to the rules, we have a number of conventions which are followed by the interpreter and programmers concerning underscores.

A name (variable, function, method) prefixed by a single underscore indicates the name is meant for internal use only in a module or class. It's not really private like other languages (Java) but merely a hint to programmers to not access the names. Additionally, the interpreter will not import the names when using a wildcard import (*from moduleA import *)* but this should be avoided in PEP008 compliant code.

A name prefixed with a double leading underscore (dunders) has its name mangled (changed) by the Python interpreter in order to avoid naming conflicts in subclasses. By default, all variables and methods are virtual in that they are inheritable and can be overridden by a subclass. Using the dunders makes it private to the specific class in which they are used and not accessible to its inherited child classes.

The upshot of these is that you should never have names of your own with both leading and trailing underscores - these should be reserved for system use. A single underscore prefix means that the name is not imported from a module, and names with two leading underscores are mangled, and so localised.
We shall be seeing examples of these conventions later.

Type specific methods

Actions on objects are done by calling *methods*

- A method is implemented as a *function* - a named code block

*object***.***method* **(**[*arg1*[,*arg2*…]]**)**

- *object* need not be a variable

Which methods may be used?

- Depends on the Class (type) of the object
- `dir(`*object*`)` lists the methods available
- `help(`*object*`)` often gives help text

Examples:

```
name.upper()            names.pop()
name.isupper()          mydict.keys()
names.count()           myfile.flush()
```

Just about everything in Python is an object, and carrying out an operation on an object, or querying an attribute, is often done by calling a function, more correctly termed a method, from the object itself. There are several advantages to this system compared to just calling a general function. For example, you know that you are operating on the correct type - there is no other way.
In the examples, we have made up some variables:

name is a string object
names is a list object
mydict is a dictionary object
myfile is a file object

The method names shown are mostly self-explanatory, but full documentation for them (and all the others) is available in the online help. The help() function prints help text (docstrings) for the class and all the methods in that class. An alternative is print (object._doc_) which displays the docstring for the class only.

Operators and type

An operator carries out an operation on an object

- Produces a result which does not (usually) alter the object
- The operation depends on the Class (type) of the object

→ List the Class using the **`type`** built-in function

```
a = 42
b = 9
print(a + b)
print(type(a))

a = 'Hello '
b = 'World!'
print(a + b)
print(type(a))
```

a and b refer to integers

```
51
<class 'int'>
```

a and b now refer to strings

```
Hello World!
<class 'str'>
```

- A list of Python operators is given after the chapter summary

py3

Python 2 reported `type` instead of `class`

An operator is usually a symbol (see the slide after the summary) which carries out an operation on a single object (unary operator) or between two operators (binary operator). The result has a value, which may be passed to a function (like print) or used on the right-hand side of an assignment.
Arithmetic operators like + (plus), - (minus), * (multiply), /(divide), and % (modulus) have familiar meanings when used with numbers, but what do they mean when used with strings?
Operators have different meanings depending on the type of object they are operating on, some of which are not at all obvious. For example, % does string formatting - something that you would not have guessed!
It is therefore, imperative that we know what type (class) of object we are dealing with, and the type function will tell us that.

Augmented assignments

A convenient shorthand for some assignments

```
stein = 1
pint  = 1
```

```
stein = stein + pint
```

```
stein += pint
```

Use any arithmetic operator

```
lhs = lhs + rhs
lhs = lhs - rhs
lhs = lhs * rhs
lhs = lhs / rhs
lhs = lhs % rhs
```

```
lhs += rhs
lhs -= rhs
lhs *= rhs
lhs /= rhs
lhs %= rhs
```

Augmented assignment is an assignment!
It has a result, which is usually ignored.

Augmented assignments are called Compound assignments in some languages and come from an ancient language now lost in the mists of the distant past (C).
The expression a += b can be read as "a is incremented by the value of b". Therefore, in the code above, the variable stein, initially at 1 is incremented by the value of pint, which is 1. After the assignment, the value of stein becomes 2 and the value of pint remains 1.
Augmented assignment operators are more succinct than the long hand approach, so programmers tend to write expressions like this:

```
total += subtotal
geometric *= progression
```

Rather than like this:

```
total = total + subtotal
geometric = geometric * progression
```

Usually, the resulting code generated will be the same. Like other operators, their meaning depends on the class of the object, for example += on a string means append.

```
a = 'Hello '
b = 'World!'
```

```
a += b
```

print(a) gives Hello World! and a new string object is created. This operation is optimised on CPython.

Python 3 types

Sequences

- Numbers

`3.142, 42, 0x3f, 0o664`

- Bytes

`b'Norwegian Blue', b"Mr. Khan's bike"`

- Strings

`'Norwegian Blue', "Mr. Khan's bike", r'C:\Numbers'`

- Tuples

Immutable

`(47, 'Spam', 'Major', 683, 'Ovine Aviation')`

- Lists

`['Cheddar', ['Camembert', 'Brie'], 'Stilton']`

Mutable

- Bytearrays

`bytearray(b'abc')`

- Dictionaries

`{'Sword':'Excalibur', 'Bird':'Unladen Swallow'}`

- Sets

`{'Chapman', 'Cleese', 'Idle', 'Jones', 'Palin'}`

Some of these types are obvious, but some require an explanation. Note the notation for octal (base 8) numbers. In old releases of Python, any number starting with a 0 (zero) would be an octal value, in Python 2.6 the prefix 0o (zero, lowercase oh) was introduced, and at Python 3 the leading zero no longer means octal. Old Python also used a trailing L to mean 'long' and a trailing 'U' to mean 'unsigned' - both are now removed.

Strings of text are objects, and once a variable of this type has been created, a method (function) can be called on it.

Numbers, strings, and Tuples are immutable, that is they cannot be altered. References to them can change to refer to different values, but the values themselves cannot. This enables Python to save space by storing just one instance of literals used in a program, regardless of how many references there are to it.

Strings, Lists and Tuples are ordered collections of objects, also known as sequences. We have a chapter on string handling later, and also discuss lists and tuples further.

Dictionaries are collections of objects accessed by key, and are similar to associative arrays in awk and PHP, and hashes in Perl and Ruby.

Sets were introduced into Python 2.4 and are described in a later chapter. There is also an immutable set: frozenset.
Lists, Tuples, Dictionaries, and Sets are known as collections, and are discussed in more detail in the Collections chapter.
A byte object is an immutable array of 8-bit values, whereas a bytearray is a mutable array of 8-bit values.

Switching types

Sometimes Python switches types automatically

```
num = 42
pi  = 3.142
num = 42/pi
print(num)
```

`num` gets automatic promotion

`13.367281986`

Sometimes you have to encourage it

- This avoids unexpected changes of type

```
print("Unused port: "  +  count)
TypeError: Can't convert 'int' object to str implicitly
```

- Use the `str()` function to return an object as a string
- Use `int()` or `float()` to return an object as a number
- Other functions available to return lists and tuples from strings

```
print("Unused port: " + str(count))
```

Like most modern languages, Python converts between numeric types automatically, generally from the narrow to the wider if there is a choice. If you actually want an integer division (so the result is truncated), then use the // operator. However, with other types things are not so clear-cut.

One form of (almost invisible) conversion is that given by the print built-in in the first example. In IDLE, if we look at the variable value by typing num, then the result is 13.367281985996181, but this is rounded by print to 13.367281986.

A common error when beginning Python is the type error shown. Python does not know if the + means string concatenation or arithmetic, anyway mixing objects of a different type is bad coding. We must explicitly tell Python what to convert and when - there is no hidden magic here.

One aid to typing is to use a naming system for your variables, for example a modified Hungarian notation which prefixes a string variable with 's', an integer with an 'i', a List with an 'l', and so on. For example: iCount, sName, lNames. This is, effectively, inventing your own sigil system.

Where we appear to call functions like str, int, and float (and tuple, list and dict which we will see latter) they are really the name of the class, so what we are really doing is constructing an object. You can find out the memory size of an object by using the `sys.getsizeof()` call, for example:

```
import sys
print("Size of count", sys.getsizeof(count), "bytes")
print("Size of str(count)",
sys.getsizeof(str(count)),"bytes")
```

Python lists introduced

Python lists are similar to arrays in other languages

- Items may be accessed from the left by an index starting at 0
- Items may be accessed from the right by an index starting at -1
- Specified as a comma-separated list of objects inside [ ]

```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg']
print(cheese[1])
cheese[-1] = 'Red Leicester'
print(cheese)
```

```
Stilton
['Cheddar', 'Stilton', 'Red Leicester']
```

Multi-dimensional lists are just lists containing others

```
cheese = ['Cheddar', ['Camembert', 'Brie'], 'Stilton']
print(cheese[1][0])
```

```
Camembert
```

Lists are objects containing a sequential collection of other objects, commonly called elements. Elements may be accessed by a position (counting from zero) specified within [ ] which is probably familiar from other languages.
Lists are Mutable, that is they may be changed, so they are similar to arrays in some languages. They are dynamic in that they may be extended or shrunk. New items may be added anywhere with the list, and also removed.
We discuss lists in more detail in the Collections chapter.

Python tuples introduced

Tuples are *immutable* (read-only) objects

- Specified as a comma-separated list of objects, often inside ( )
  - → Can be specified inside ( ) sometimes required for precedence
  - → The comma makes a tuple, not the ( )
- Can be indexed in the same way as lists
  - → Starting from 0 on the left or -1 on the right

```
mytuple = 'eggs', 'bacon', 'spam', 'tea'
print(mytuple)
print(mytuple[1])
print(mytuple[-1])
```

```
('eggs', 'bacon', 'spam', 'tea')
bacon
tea
```

- Can be reassigned, but not altered

```
mytuple[2] = 'John'
```

```
TypeError: 'tuple' object does not support item assignment
```

Tuples are Immutable (read only), for example if you attempt to append:

```
TupleVar.append('Vikings')
Traceback (most recent call last):
  File "liststuples.py", line 6, in ?
    TupleVar.append('Vikings')
AttributeError: 'tuple' object has no attribute 'append'
```

Tuples elements can be assigned, provided they are other variables. It may seem strange that Python has two seemingly similar types, tuples and lists. While lists are more flexible than tuples, there is a penalty to pay in performance overhead. In most cases, where either could be used, tuples are most efficient than lists.

Although parentheses are often associated with tuples, these are usually optional. So in the example on the slide:

```
mytuple = ('eggs', 'bacon', 'spam', 'tea')
```

is equally valid, and will produce the same result.

We discuss tuples further in the Collections chapter.

Python dictionaries introduced

A Dictionary object is an unordered collection of objects

- Constructed from { }

  varname **=** **{***key1***:***object1***,***key2***:***object2***,***key3***:***object3***,…}**
- Accessed by key

→ A key is a text string, or anything that yields a text string

  *varname***[***key***]** **=** *object*

```
mydict = {'Australia':'Canberra', 'Eire':'Dublin',
          'France':'Paris', 'Finland':'Helsinki',
          'UK':'London', 'US':'Washington'
         }
print(mydict['UK'])

country = 'Iceland'
mydict[country] = 'Reykjavik'
```

```
London
```

Dictionaries are just like associative arrays in awk and PHP, or hashes in Perl and Ruby. They are constructed from lists of key:object pairs, inside braces (curly brackets), although you may also assign them from the dict function, for example:

mydict = dict(Sweden = 'Stockholm', Norway = 'Oslo')

The key is a text string, while the value is an object of any valid class, including a list, tuple, or dictionary. No special syntax is required to access them.

Dictionary key:value pairs are not ordered, which makes them different to PHP associative arrays and Ruby hashes. A list of the keys may be extracted using the keys() method, and values with the values() method.

We discuss dictionaries, and their related type sets, in the Collections chapter.

SUMMARY

- **A Python variable is a reference to an object**
- **Python variable names are case-sensitive**
  - Watch out for leading underscores
- **Variables are accessed using operators and methods**
  - `dir(`*object*`)` lists the methods available
- **Lists are like arrays in other languages**
- **Tuples are "immutable"**
  - But can contain variables
- **Dictionaries store objects accessed by key**
  - Keys are unique
  - Not ordered

Python operators

`or`	**logical OR**
`and`	**logical AND**
`not`	**logical NOT**
`< <= > >=`	**comparison operators**
`== !=`	**equality operators**
`is`	**object identity test**
`in`	**object membership test**
`| ^`	**binary OR, XOR**
`&`	**binary AND**
`<< >>`	**binary shift**
`- +`	**subtract, add**
`* / // %`	**multiply, divide, integer-divide, modulo**
`@`	**matrix multiplication (3.5)**
`~ **`	**complement, exponentiation**
`await`	**await expression (3.5)**

The operators listed in reverse order of precedence (or is the lowest precedence).

Difference between / and // is best shown as an example:

x = 2	
y = x/3	gives 0.666666666667
y = x//3	gives 0

`Await` is used with coroutines and becomes a keyword in 3.7, along with async. Both were introduced at 3.5 and require the asyncio module. This module is current provisional and may include changes that are not backward compatible.

The `@` operator (`matmul`) is intended for matrix multiplication and has the same precedence as multiplication. No built-in types currently support this operator, it is intended for third-party modules.

Python reserved words

The following are illegal as variable or function names in Python

False	None	True	and	as*	assert
async^	await^	break	class	continue	def
del	elif	else	except	exec~	finally
for	from	global	if	import	in
is	lambda	nonlocal+	not	or	pass
raise	return	try	while	with*	yield

* version 2.6 and later
\+ version 3.0
~ not in version 3.0
^ version 3.7

exec and **print** were keywords prior to 3.0, now they are built-in functions

py3

Briefly, the meanings of these reserved words are:

and, not, or logical operators
assert, raise trigger an exception
async, await used with async coroutines
(from 3.5, reserved at 3.7)
break exit the current loop
class create a class
object
continue do the next
iteration of the current loop
def create a function
object
del delete an item
from a list
except indicates an
exception handler for a try block
exec execute code
finally statements always executed
after a try block
for sequence iteration

loops
from used with 'import'
to specify imported names
global declare variable as
global
if, else, elif conditional clauses
import find and load a
module
in tests sequence
membership
is identity test
lambda create an
anonymous function
pass empty statement
(no-op)
print write to stdout,
appending a "\n"
return return a value
from a function
try catches
exceptions
while conditional loop statement
yield is an extension, with and as are reserved words in 2.6,
nonlocal in 3.0