Python is an interpreted language. The Python interpreter runs a program by executing one statement at a time

The >>> you see is the prompt where you'll type code expressions. To exit the Python interpreter and return to the command prompt, you can either type exit() or press Ctrl-D.

While some Python programmers execute all of their Python code in this way, those doing data analysis or scientific computing make use of IPython, an enhanced Python interpreter, or Jupyter notebooks, web-based code notebooks originally created within the IPython project

When you use the %run command, IPython executes the code in the specified file in the same process, enabling you to explore the results interactively when it's done:

In [1]:
```
%run helloWorld.py
```

```
Hello World!
```

The default IPython prompt adopts the numbered In [2]: style compared with the standard >>> prompt.

2.2 IPython Basics

In [2]:
```
import numpy as np
data = {i : np.random.randn() for i in range(7)}
data
```

Out[2]:
```
{0: 0.50642062205232,
 1: 0.30585987205084436,
 2: -1.3006870723234079,
 3: -2.521900649728477,
 4: -1.2215480057196537,
 5: 0.3118194481848336,
 6: 0.19271037365402097}
```

Many kinds of Python objects are formatted to be more readable, or pretty-printed, which is distinct from normal printing with print. If you printed the above data variable in the standard Python interpreter, it would be much less readable:

In [3]:
```
from numpy.random import randn
>>> data = {i : randn() for i in range(7)}
>>> print(data)
```

```
{0: 0.34133483558044375, 1: 1.3009475642657342, 2: -1.1053923496123044, 3: -2.055559
9676956913, 4: 1.8089717319378391, 5: -1.6550543716030084, 6: 0.000698966289628906}
```

Running the Jupyter Notebook

One of the major components of the Jupyter project is the notebook, a type of interactive document for code, text (with or without markup), data visualizations, and other output.

The Jupyter notebook interacts with kernels, which are implementations of the Jupyter interactive computing protocol in any number of programming lan- guages. Python's Jupyter kernel uses the IPython system for its underlying behavior. To start up Jupyter, run the command jupyter notebook in a terminal.

On many platforms, Jupyter will automatically open up in your default web browser (unless you start it with --no-browser). Otherwise, you can navigate to the HTTP address printed when you started the notebook, here http://localhost:8888/

Many people use Jupyter as a local computing environment, but it can also be deployed on servers and accessed remotely

Tab Completion

While entering expressions in the shell, pressing the Tab key will search the namespace for any variables (objects, functions, etc.) matching the characters you have typed so far

```
In [4]: an_apple = 27
```

```
In [5]: an_example = 42
```

```
In [6]: # an <press tab>
```

Naturally, you can also complete methods and attributes on any object after typing a period:

```
In [7]: b = [1, 2, 3]
```

```
In [8]: #b.<press tab>
```

The same goes for modules:

```
In [1]: import datetime
```

```
In [10]: #datetime. <press tab>
```

In the Jupyter notebook and newer versions of IPython (5.0 and higher), the auto-completions show up in a drop-down box rather than as text output.

Note that IPython by default hides methods and attributes starting with underscores, such as magic methods and internal "private" methods and attributes, in order to avoid cluttering the display (and confusing novice users!). These, too, can be tab-completed, but you must first type an underscore to see them. If you prefer to always see such methods in tab

completion, you can change this setting in the IPython configuration. See the IPython documentation to find out how to do this

Tab completion works in many contexts outside of searching the interactive namespace and completing object or module attributes. When typing anything that looks like a file path (even in a Python string), pressing the Tab key will complete anything on your computer's filesystem matching what you've typed:

In [11]:
```
#datasets/movielens/<press Tab>
```

In [12]:
```
#'datasets/movielens/<press Tab>
```

Combined with the %run command (see "The %run Command" on page 25), this functionality can save you many keystrokes. Another area where tab completion saves time is in the completion of function keyword arguments (and including the = sign!).

Introspection

Using a question mark (?) before or after a variable will display some general information about the object:

In [2]:
```
b = [1, 2, 3]
```

In [3]:
```
b?
```

In [15]:
```
print?
```

This is referred to as object introspection. If the object is a function or instance method, the docstring, if defined, will also be shown. Suppose we'd written the following function (which you can reproduce in IPython or Jupyter):

In [16]:
```
def add_numbers(a, b):
 """
 Add two numbers together
 Returns
 -------
 the_sum : type of arguments
 """
 return a + b
```

In [17]:
```
add_numbers?
```

Using ?? will also show the function's source code if possible:

In [18]:
```
add_numbers??
```

? has a final usage, which is for searching the IPython namespace in a manner similar to the standard Unix or Windows command line. A number of characters combined with the wildcard (*) will show all names matching the wildcard expression. For example, we could get a list of all functions in the top-level NumPy namespace containing load:

In [5]:
```
np.*load*?
```

The %run Command

You can run any file as a Python program inside the environment of your IPython session using the %run command

In [6]:
```
%run ipython_script_test.py
```

The script is run in an empty namespace (with no imports or other variables defined) so that the behavior should be identical to running the program on the command line using python script.py. All of the variables (imports, functions, and globals) defined in the file (up until an exception, if any, is raised) will then be accessible in the IPython shell:

In [7]:
```
c
```

Out[7]:
```
7.5
```

In [8]:
```
result
```

Out[8]:
```
1.4666666666666666
```

If a Python script expects command-line arguments (to be found in sys.argv), these can be passed after the file path as though run on the command line.

Should you wish to give a script access to variables already defined in the interactive IPython namespace, use %run -i instead of plain %run

In the Jupyter notebook, you may also use the related %load magic function, which imports a script into a code cell

In [23]:
```
# %load ipython_script_test.py
```

Interrupting running code

Pressing Ctrl-C while any code is running, whether a script through %run or a longrunning command, will cause a KeyboardInterrupt to be raised. This will cause nearly all Python programs to stop immediately except in certain unusual cases.

When a piece of Python code has called into some compiled extension modules, pressing Ctrl-C will not always cause the program execution to stop immediately. In such cases, you will have to either wait until control is returned to the Python interpreter, or in more dire circumstances, forcibly terminate the Python process.

Executing Code from the Clipboard

The most foolproof methods are the %paste and %cpaste magic functions. %paste takes whatever text is in the clipboard and executes it as a single block in the shell

In [9]:
```
# copy some code and open the cmd and type ipython and then type the below command
# %paste
```

With the %cpaste block, you have the freedom to paste as much code as you like before executing it. You might decide to use %cpaste in order to look at the pasted code before executing it. If you accidentally paste the wrong code, you can break out of the %cpaste prompt by pressing Ctrl-C

Terminal Keyboard Shortcuts

In [25]:
```
#first type ipython in cmd then use the below commands

# Ctrl-P or up-arrow Search backward in command history for commands starting with
# Ctrl-N or down-arrow Search forward in command history for commands starting with
# Ctrl-R Readline-style reverse history search (partial matching)
# Ctrl-Shift-V Paste text from clipboard
# Ctrl-C Interrupt currently executing code
# Ctrl-A Move cursor to beginning of line
# Ctrl-E Move cursor to end of line
# Ctrl-K Delete text from cursor until end of line
# Ctrl-U Discard all text on current line
# Ctrl-F Move cursor forward one character
# Ctrl-B Move cursor back one character
# Ctrl-L Clear screen
```

About Magic Commands

IPython's special commands (which are not built into Python itself) are known as "magic" commands

A magic command is any command prefixed by the percent symbol %. For example, you can check the execution time of any Python statement, such as a matrix multiplication, using the %timeit magic function (which will be discussed in more detail later):

In [26]:
```
a = np.random.randn(100, 100)
```

In [27]:
```
a.shape
```

Out[27]:
```
(100, 100)
```

In [28]:
```
%timeit np.dot(a, a)
```

```
99.1 µs ± 7.41 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Magic commands can be viewed as command-line programs to be run within the IPython system. Many of them have additional "command-line" options, which can all be viewed (as you might expect) using ?

In [10]:
```
%debug?
```

Magic functions can be used by default without the percent sign, as long as no variable is defined with the same name as the magic function in question. This feature is called automagic and can be enabled or disabled with %automagic.

Some magic functions behave like Python functions and their output can be assigned to a variable:

In [30]:
```
%pwd
```

Out[30]:
```
'C:\\Users\\ankit19.gupta\\OneDrive - Reliance Corporate IT Park Limited\\Desktop
\\Practice_Code\\Python_Practice\\Python_For_Data_Analysis'
```

In [31]:
```
foo = %pwd
foo
```

Out[31]:
```
'C:\\Users\\ankit19.gupta\\OneDrive - Reliance Corporate IT Park Limited\\Desktop
\\Practice_Code\\Python_Practice\\Python_For_Data_Analysis'
```

In [11]:
```
%quickref
```

In [33]:
```
%magic
```

In [34]:
```
# Command ----> Description
# %quickref ----> Display the IPython Quick Reference Card
# %magic ----> Display detailed documentation for all of the available magic comman
# %debug ----> Enter the interactive debugger at the bottom of the last exception t
# %hist ----> Print command input (and optionally output) history
# %pdb ----> Automatically enter debugger after any exception
# %paste ----> Execute preformatted Python code from clipboard
# %cpaste ----> Open a special prompt for manually pasting Python code to be execut
# %reset ----> Delete all variables/names defined in interactive namespace
# %page OBJECT ----> Pretty-print the object and display it through a pager
# %run script.py ----> Run a Python script inside IPython
# %prun statement ----> Execute statement with cProfile and report the profiler out
# %time statement ----> Report the execution time of a single statement
# %timeit statement ----> Run a statement multiple times to compute an ensemble ave
```

```
# timing code with very short execution time
# %who, %who_ls, %whos ----> Display variables defined in interactive namespace, wi
# verbosity
# %xdel variable ----> Delete a variable and attempt to clear any references to the
```

Matplotlib Integration

One reason for IPython's popularity in analytical computing is that it integrates well with data visualization and other user interface libraries like matplotlib.

The %matplotlib magic function configures its integration with the IPython shell or Jupyter notebook. This is important, as otherwise plots you create will either not appear (notebook) or take control of the session until closed (shell).

In the IPython shell, running %matplotlib sets up the integration so you can create multiple plot windows without interfering with the console session

In [35]:
```
%matplotlib
import matplotlib.pyplot as plt
plt.plot(np.random.randn(50).cumsum())
```

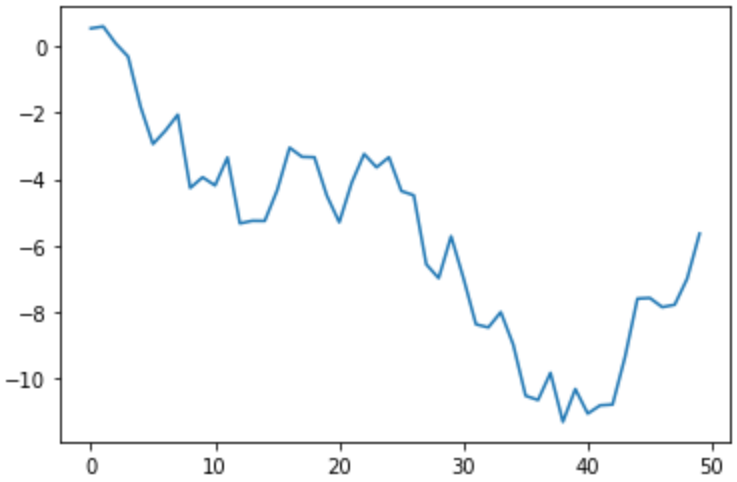
```
Using matplotlib backend: TkAgg
```

Out[35]:
```
[<matplotlib.lines.Line2D at 0x2067ac8f470>]
```

In Jupyter, the command is a little different

In [36]:
```
%matplotlib inline
import matplotlib.pyplot as plt
plt.plot(np.random.randn(50).cumsum())
```

Out[36]:
```
[<matplotlib.lines.Line2D at 0x2067b1d15c0>]
```



2.3 Python Language Basics

Indentation

I strongly recommend using four spaces as your default indentation and replacing tabs with four spaces. Many text editors have a setting that will replace tab stops with spaces automatically (do this!). Some people use tabs or a different number of spaces, with two spaces not being terribly uncommon. By and large, four spaces is the standard adopted by the vast majority of Python programmers, so I recommend doing that in the absence of a compelling reason otherwise.

Putting multiple statements on one line is generally discouraged in Python as it often makes code less readable

Everything is an object

An important characteristic of the Python language is the consistency of its object model. Every number, string, data structure, function, class, module, and so on exists in the Python interpreter in its own "box," which is referred to as a Python object. Each object has an associated type (e.g., string or function) and internal data. In practice this makes the language very flexible, as even functions can be treated like any other object

Almost every object in Python has attached functions, known as methods, that have access to the object's internal contents. You can call them using the following syntax:
obj.some_method(x, y, z)

Almost every object in Python has attached functions, known as methods, that have access to the object's internal contents. You can call them using the following syntax:
obj.some_method(x, y, z)

Functions can take both positional and keyword arguments:
result = f(a, b, c, d=5, e='foo')

Variables and argument passing

When assigning a variable (or name) in Python, you are creating a reference to the object on the righthand side of the equals sign. In practical terms, consider a list of integers

In [1]:
```
a = [1, 2, 3]
```

Suppose we assign a to a new variable b:

In [2]:
```
b = a
```

In some languages, this assignment would cause the data [1, 2, 3] to be copied. In Python, a and b actually now refer to the same object, the original list [1, 2, 3]

You can prove this to yourself by appending an element to a and then examining b:

In [3]:
```
a.append(4)
```

In [4]:
```
a
```

Out[4]:
```
[1, 2, 3, 4]
```

In [5]:
```
b
```

Out[5]:
```
[1, 2, 3, 4]
```

Assignment is also referred to as binding, as we are binding a name to an object. Variable names that have been assigned may occasion- ally be referred to as bound variables

When you pass objects as arguments to a function, new local variables are created referencing the original objects without any copying. If you bind a new object to a vari- able inside a function, that change will not be reflected in the parent scope. It is therefore possible to alter the internals of a mutable argument. Suppose we had the following function:

In [6]:
```
def append_element(some_list, element):
 some_list.append(element)
```

In [7]:
```
data = [1, 2, 3]
```

In [8]:
```
append_element(data, 4)
data
```

Out[8]:
```
[1, 2, 3, 4]
```

Dynamic references, strong types

In contrast with many compiled languages, such as Java and C++, object references in Python have no type associated with them. There is no problem with the following:

In [9]:
```
a = 5
type(a)
```

Out[9]:
```
int
```

In [10]:
```
a = 'foo'
type(a)
```

Out[10]:
```
str
```

Variables are names for objects within a particular namespace; the type information is stored in the object itself. Some observers might hastily conclude that Python is not a “typed

language." This is not true; consider this example:

In [11]:
```
'5' + 5
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-11-4dd8efb5fac1> in <module>
----> 1 '5' + 5

TypeError: must be str, not int
```

In some languages, such as Visual Basic, the string '5' might get implicitly converted (or casted) to an integer, thus yielding 10. Yet in other languages, such as JavaScript, the integer 5 might be casted to a string, yielding the concatenated string '55'. In this regard Python is considered a strongly typed language, which means that every object has a specific type (or class), and implicit conversions will occur only in certain obvi- ous circumstances, such as the following:

In [12]:
```
a = 4.5
b = 2
print('a is {0}, b is {1}'.format(type(a), type(b)))
```

```
a is <class 'float'>, b is <class 'int'>
```

In [13]:
```
a / b
```

Out[13]:
```
2.25
```

You can check that an object is an instance of a particular type using the isinstance function:

In [14]:
```
a = 5
isinstance(a, int)
```

Out[14]:
```
True
```

isinstance can accept a tuple of types if you want to check that an object's type is among those present in the tuple:

In [16]:
```
a = 5; b = 4.5
isinstance(a, (int, float))
```

Out[16]:
```
True
```

In [17]:
```
isinstance(b, (int, float))
```

Out[17]:
```
True
```

Attributes and methods

Objects in Python typically have both attributes (other Python objects stored "inside" the object) and methods (functions associated with an object that can have access to the

object's internal data). Both of them are accessed via the syntax obj.attribute_name:

In [18]:
```
a = 'foo'
```

In [19]:
```
#a.<Press Tab>
```

Attributes and methods can also be accessed by name via the getattr function:

In [20]:
```
getattr(a, 'split')
```

Out[20]:
```
<function str.split>
```

In other languages, accessing objects by name is often referred to as "reflection." While we will not extensively use the functions getattr and related functions hasattr and setattr in this book, they can be used very effectively to write generic, reusable code

Duck typing

Often you may not care about the type of an object but rather only whether it has certain methods or behavior. This is sometimes called "duck typing," after the saying "If it walks like a duck and quacks like a duck, then it's a duck." For example, you can verify that an object is iterable if it implemented the iterator protocol. For many objects, this means it has a __iter__ "magic method," though an alternative and better way to check is to try using the iter function:

In [22]:
```
def isiterable(obj):
    try:
        iter(obj)
        return True
    except TypeError: # not iterable
        return False
```

This function would return True for strings as well as most Python collection types:

In [23]:
```
isiterable('a string')
```

Out[23]:
```
True
```

In [24]:
```
isiterable([1, 2, 3])
```

Out[24]:
```
True
```

In [25]:
```
iter([1, 2, 3])
```

Out[25]:
```
<list_iterator at 0x29130400160>
```

In [26]:
```
isiterable(5)
```

Out[26]:
```
False
```

A place where I use this functionality all the time is to write functions that can accept multiple kinds of input. A common case is writing a function that can accept any kind of sequence (list, tuple, ndarray) or even an iterator. You can first check if the object is a list (or a NumPy array) and, if it is not, convert it to be one:

In [29]:
```
x='ankit'
if not isinstance(x, list) and isiterable(x):
 x = list(x)
x
```

Out[29]:
```
['a', 'n', 'k', 'i', 't']
```

Imports

In Python a module is simply a file with the .py extension containing Python code. Suppose that we had the following module:

In [30]:
```
# some_module.py
PI = 3.14159
def f(x):
 return x + 2
def g(a, b):
 return a + b
```

If we wanted to access the variables and functions defined in some_module.py, from another file in the same directory we could do:

In [31]:
```
import some_module
result = some_module.f(5)
pi = some_module.PI
pi
```

Out[31]:
```
3.14159
```

In [32]:
```
result
```

Out[32]:
```
7
```

Or equivalently:

In [33]:
```
from some_module import f, g, PI
result = g(5, PI)
result
```

Out[33]:
```
8.14159
```

By using the as keyword you can give imports different variable names:

In [34]:
```
import some_module as sm
from some_module import PI as pi, g as gf
r1 = sm.f(pi)
r2 = gf(6, pi)
```

In [35]:
```
r1
```

Out[35]: 5.14159

In [36]:
```
r2
```

Out[36]: 9.14159

To check if two references refer to the same object, use the is keyword. is not is also perfectly valid if you want to check that two objects are not the same:

In [37]:
```
a = [1, 2, 3]
b = a
c = list(a)
a is b
```

Out[37]: True

In [38]:
```
a is not c
```

Out[38]: True

Since list always creates a new Python list (i.e., a copy), we can be sure that c is distinct from a. Comparing with is is not the same as the == operator, because in this case we have:

In [39]:
```
a == c
```

Out[39]: True

A very common use of is and is not is to check if a variable is None, since there is only one instance of None:

In [40]:
```
a = None
a is None
```

Out[40]: True

In [41]:
```
# ##Binary operators
# Operation --> Description
# a + b --> Add a and b
# a - b --> Subtract b from a
# a * b --> Multiply a by b
# a / b --> Divide a by b
# a // b --> Floor-divide a by b, dropping any fractional remainder
# a ** b --> Raise a to the b power
# a & b --> True if both a and b are True; for integers, take the bitwise AND
```

```
# a | b --> True if either a or b is True; for integers, take the bitwise OR
# a ^ b --> For booleans, True if a or b is True, but not both; for integers, take
# a == b -->  True if a equals b
# a != b --> True if a is not equal to b
# a <= b, a < b --> True if a is less than (less than or equal) to b
# a > b, a >= b --> True if a is greater than (greater than or equal) to b
# a is b --> True if a and b reference the same Python object
# a is not b --> True if a and b reference different Python objects
```

Mutable and immutable objects

Most objects in Python, such as lists, dicts, NumPy arrays, and most user-defined types (classes), are mutable. This means that the object or values that they contain can be modified:

In [42]:
```
a_list = ['foo', 2, [4, 5]]
a_list[2] = (3, 4)
a_list
```

Out[42]:
```
['foo', 2, (3, 4)]
```

Others, like strings and tuples, are immutable:

In [43]:
```
a_tuple = (3, 5, (4, 5))
a_tuple[1] = 'four'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-43-2c9bddc8679c> in <module>
      1 a_tuple = (3, 5, (4, 5))
----> 2 a_tuple[1] = 'four'

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

Remember that just because you can mutate an object does not mean that you always should. Such actions are known as side effects. For example, when writing a function, any side effects should be explicitly communicated to the user in the function's documentation or comments. If possible, I recommend trying to avoid side effects and favor immutability, even though there may be mutable objects involved.

Scalar Types

Python along with its standard library has a small set of built-in types for handling numerical data, strings, boolean (True or False) values, and dates and time. These "single value" types are sometimes called scalar types and we refer to them in this book as scalars. See Table 2-4 for a list of the main scalar types. Date and time handling will be discussed separately, as these are provided by the datetime module in the standard library.

In [44]:
```
# # Table 2-4. Standard Python scalar types
# Type --> Description
# None --> The Python “null” value (only one instance of the None object exists)
# str --> String type; holds Unicode (UTF-8 encoded) strings
# bytes --> Raw ASCII bytes (or Unicode encoded as bytes)
# float --> Double-precision (64-bit) floating-point number (note there is no separa
# bool --> A True or False value
# int --> Arbitrary precision signed integer
```

Floating-point numbers are represented with the Python float type. Under the hood each one is a double-precision (64-bit) value. They can also be expressed with scientific notation:

In [45]:
```
fval = 7.243
fval2 = 6.78e-5
fval2
```

Out[45]:
```
6.78e-05
```

Integer division not resulting in a whole number will always yield a floating-point number

In [46]:
```
3 / 2
```

Out[46]:
```
1.5
```

To get C-style integer division (which drops the fractional part if the result is not a whole number), use the floor division operator //:

In [47]:
```
3 // 2
```

Out[47]:
```
1
```

For multiline strings with line breaks, you can use triple quotes, either ''' or """:

In [48]:
```
c = """
This is a longer string that
spans multiple lines
"""
c
```

Out[48]:
```
'\nThis is a longer string that\nspans multiple lines\n'
```

It may surprise you that this string c actually contains four lines of text; the line breaks after """ and after lines are included in the string. We can count the new line characters with the count method on c:

In [49]:
```
c.count('\n')
```

Out[49]:
```
3
```

Python strings are immutable; you cannot modify a string:

In [50]:
```
a = 'this is a string'
a[10] = 'f'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-50-cfa170a67205> in <module>
      1 a = 'this is a string'
----> 2 a[10] = 'f'

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

In [51]:
```
b = a.replace('string', 'longer string')
b
```

Out[51]:
```
'this is a longer string'
```

Afer this operation, the variable a is unmodified:

In [52]:
```
a
```

Out[52]:
```
'this is a string'
```

Many Python objects can be converted to a string using the str function

In [53]:
```
a = 5.6
str(a)
```

Out[53]:
```
'5.6'
```

Strings are a sequence of Unicode characters and therefore can be treated like other sequences, such as lists and tuples

In [54]:
```
s = 'python'
list(s)
```

Out[54]:
```
['p', 'y', 't', 'h', 'o', 'n']
```

In [55]:
```
s[:3]
```

Out[55]:
```
'pyt'
```

The syntax s[:3] is called slicing and is implemented for many kinds of Python sequences.

The backslash character \ is an escape character, meaning that it is used to specify special characters like newline \n or Unicode characters. To write a string literal with backslashes, you need to escape them:

In [60]:
```
s = '12\\34'
print(s)
```

```
12\34
```

If you have a string with a lot of backslashes and no special characters, you might find this a bit annoying. Fortunately you can preface the leading quote of the string with r, which means that the characters should be interpreted as is:

In [61]:
```
s = r'this\has\no\special\characters'
s
```

Out[61]:
```
'this\\has\\no\\special\\characters'
```

The r stands for raw

Adding two strings together concatenates them and produces a new string:

In [62]:
```
a = 'this is the first half '
b = 'and this is the second half'
a + b
```

Out[62]:
```
'this is the first half and this is the second half'
```

String templating or formatting is another important topic. The number of ways to do so has expanded with the advent of Python 3, and here I will briefly describe the mechanics of one of the main interfaces. String objects have a format method that can be used to substitute formatted arguments into the string, producing a new string

In [63]:
```
template = '{0:.2f} {1:s} are worth US${2:d}'
```

In [64]:
```
template
```

Out[64]:
```
'{0:.2f} {1:s} are worth US${2:d}'
```

In this string,
• {0:.2f} means to format the first argument as a floating-point number with two decimal places.
• {1:s} means to format the second argument as a string.
• {2:d} means to format the third argument as an exact integer.

To substitute arguments for these format parameters, we pass a sequence of arguments to the format method:

In [66]:
```
template.format(4.5560, 'Argentine Pesos', 1)
```

Out[66]:
```
'4.56 Argentine Pesos are worth US$1'
```

Bytes and Unicode

In modern Python (i.e., Python 3.0 and up), Unicode has become the first-class string type to enable more consistent handling of ASCII and non-ASCII text. In older versions of Python,

strings were all bytes without any explicit Unicode encoding. You could convert to Unicode assuming you knew the character encoding. Let's look at an example:

```
In [67]: val = "español"
```

```
In [68]: val
```

```
Out[68]: 'español'
```

We can convert this Unicode string to its UTF-8 bytes representation using the encode method:

```
In [69]: val_utf8 = val.encode('utf-8')
```

```
In [70]: val_utf8
```

```
Out[70]: b'espa\xc3\xb1ol'
```

```
In [71]: type(val_utf8)
```

```
Out[71]: bytes
```

Assuming you know the Unicode encoding of a bytes object, you can go back using the decode method:

```
In [72]: val_utf8.decode('utf-8')
```

```
Out[72]: 'español'
```

While it's become preferred to use UTF-8 for any encoding, for historical reasons you may encounter data in any number of different encodings:

```
In [73]: val.encode('latin1')
```

```
Out[73]: b'espa\xf1ol'
```

```
In [74]: val.encode('utf-16')
```

```
Out[74]: b'\xff\xfee\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'
```

```
In [75]: val.encode('utf-16le')
```

```
Out[75]: b'e\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'
```

It is most common to encounter bytes objects in the context of working with files, where implicitly decoding all data to Unicode strings may not be desired. Though you may seldom need to do so, you can define your own byte literals by prefixing a string with b:

In [76]:
```
bytes_val = b'this is bytes'
bytes_val
```

Out[76]:
```
b'this is bytes'
```

In [77]:
```
decoded = bytes_val.decode('utf8')
decoded # this is str (Unicode) now
```

Out[77]:
```
'this is bytes'
```

Type casting

The str, bool, int, and float types are also functions that can be used to cast values to those types:

In [1]:
```
s = '3.14159'
fval = float(s)
type(fval)
```

Out[1]:
```
float
```

In [2]:
```
int(fval)
```

Out[2]:
```
3
```

In [3]:
```
bool(fval)
```

Out[3]:
```
True
```

In [4]:
```
bool(0)
```

Out[4]:
```
False
```

None

None is the Python null value type. If a function does not explicitly return a value, it implicitly returns None:

In [5]:
```
a = None
a is None
```

Out[5]:
```
True
```

In [6]:
```
b = 5
b is not None
```

Out[6]:
```
True
```

None is also a common default value for function arguments:

In [7]:
```
def add_and_maybe_multiply(a, b, c=None):
    result = a + b
    if c is not None:
        result = result * c
    return result
```

In [8]:
```
add_and_maybe_multiply(a=2, b=3, c=4)
```

Out[8]:
```
20
```

While a technical point, it's worth bearing in mind that None is not only a reserved keyword but also a unique instance of NoneType:

In [9]:
```
type(None)
```

Out[9]:
```
NoneType
```

Dates and times

The built-in Python datetime module provides datetime, date, and time types. The datetime type, as you may imagine, combines the information stored in date and time and is the most commonly used:

In [10]:
```
from datetime import datetime, date, time
dt = datetime(2011, 10, 29, 20, 30, 21)
print(dt.day)
print(dt.minute)
```

```
29
30
```

Given a datetime instance, you can extract the equivalent date and time objects by calling methods on the datetime of the same name:

In [11]:
```
dt.date()
```

Out[11]:
```
datetime.date(2011, 10, 29)
```

In [12]:
```
dt.time()
```

Out[12]:
```
datetime.time(20, 30, 21)
```

The strftime method formats a datetime as a string:

In [13]:
```
dt.strftime('%m/%d/%Y %H:%M')
```

Out[13]:
```
'10/29/2011 20:30'
```

Strings can be converted (parsed) into datetime objects with the strptime function:

In [14]:
```
datetime.strptime('20091031', '%Y%m%d')
```

Out[14]:
```
datetime.datetime(2009, 10, 31, 0, 0)
```

When you are aggregating or otherwise grouping time series data, it will occasionally be useful to replace time fields of a series of datetimes—for example, replacing the minute and second fields with zero:

In [15]:
```
dt.replace(minute=0, second=0)
```

Out[15]:
```
datetime.datetime(2011, 10, 29, 20, 0)
```

Since datetime.datetime is an immutable type, methods like these always produce new objects.

The difference of two datetime objects produces a datetime.timedelta type:

In [16]:
```
dt2 = datetime(2011, 11, 15, 22, 30)
```

In [17]:
```
delta = dt2 - dt
```

In [18]:
```
delta
```

Out[18]:
```
datetime.timedelta(17, 7179)
```

In [19]:
```
type(delta)
```

Out[19]:
```
datetime.timedelta
```

The output timedelta(17, 7179) indicates that the timedelta encodes an offset of 17 days and 7,179 seconds.

Adding a timedelta to a datetime produces a new shifted datetime:

In [20]:
```
dt
```

Out[20]:
```
datetime.datetime(2011, 10, 29, 20, 30, 21)
```

In [21]:
```
dt + delta
```

Out[21]:
```
datetime.datetime(2011, 11, 15, 22, 30)
```

In [22]:
```
# ## Table 2-5. Datetime format specification (ISO C89 compatible)
# Type --> Description
# %Y --> Four-digit year
# %y --> Two-digit year
# %m --> Two-digit month [01, 12]
# %d --> Two-digit day [01, 31]
# %H --> Hour (24-hour clock) [00, 23]
# %I --> Hour (12-hour clock) [01, 12]
```

```
# %M --> Two-digit minute [00, 59]
# %S --> Second [00, 61] (seconds 60, 61 account for leap seconds)
# %w --> Weekday as integer [0 (Sunday), 6]
# %U --> Week number of the year [00, 53]; Sunday is considered the first day of th
# the year are “week 0”
# %W --> Week number of the year [00, 53]; Monday is considered the first day of th
# the year are “week 0”
# %z --> UTC time zone offset as +HHMM or -HHMM; empty if time zone naive
# %F --> Shortcut for %Y-%m-%d (e.g., 2012-4-18)
# %D --> Shortcut for %m/%d/%y (e.g., 04/18/12)
```

In [23]:
```
4 > 3 > 2 > 1
```

Out[23]: True

for loops

for loops are for iterating over a collection (like a list or tuple) or an iterater. The standard syntax for a for loop is:
for value in collection:
do something with value

You can advance a for loop to the next iteration, skipping the remainder of the block, using the continue keyword. Consider this code, which sums up integers in a list and skips None values:

In [25]:
```
sequence = [1, 2, None, 4, None, 5]
total = 0
for value in sequence:
 if value is None:
     continue
 total += value
```

In [26]:
```
total
```

Out[26]: 12

A for loop can be exited altogether with the break keyword. This code sums elements of the list until a 5 is reached:

In [27]:
```
sequence = [1, 2, 0, 4, 6, 5, 2, 1]
total_until_5 = 0
for value in sequence:
 if value == 5:
     break
 total_until_5 += value
```

In [28]:
```
total_until_5
```

Out[28]: 13

The break keyword only terminates the innermost for loop; any outer for loops will continue to run:

pass

pass is the "no-op" statement in Python. It can be used in blocks where no action is to be taken (or as a placeholder for code not yet implemented); it is only required because Python uses whitespace to delimit blocks

In [30]:
```
x=-3
if x < 0:
 print('negative!')
elif x == 0:
 # TODO: put something smart here
 pass
else:
 print('positive!')
```

```
negative!
```

range

The range function returns an iterator that yields a sequence of evenly spaced integers:

In [31]:
```
range(10)
```

Out[31]:
```
range(0, 10)
```

In [32]:
```
list(range(10))
```

Out[32]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Both a start, end, and step (which may be negative) can be given

In [34]:
```
list(range(0, 20, 2))
```

Out[34]:
```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

In [35]:
```
list(range(5, 0, -1))
```

Out[35]:
```
[5, 4, 3, 2, 1]
```

As you can see, range produces integers up to but not including the endpoint. A common use of range is for iterating through sequences by index:

In [36]:
```
seq = [1, 2, 3, 4]
for i in range(len(seq)):
 val = seq[i]
val
```

Out[36]: 4

Ternary expressions

A ternary expression in Python allows you to combine an if-else block that produces a value into a single line or expression. The syntax for this in Python is:
value = true-expr if condition else false-expr

In [38]:
```
x = 5
'Non-negative' if x >= 0 else 'Negative'
```

Out[38]: 'Non-negative'

In [ ]: