Quick Python on macOS

Where to find Python

On macOS, Python has come bundled with the OS for as long as I can remember. Apple's Python is in `/usr/bin/python3`. Note: no space between python and 3. From Terminal.app:

```
> /usr/bin/python3
Python 3.9.6 (default, Oct 18 2022, 12:41:40)
[Clang 14.0.0 (clang-1400.0.29.202)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

It should be available by simply typing `python3` at the prompt.

The advantage of this is there is nothing to install. The disadvantage is that Apple's Python is *theirs*. If you want your own Python, I recommend Homebrew. On my machine

```
> which python3
/usr/local/bin/python3
>
```

which is the location used by Homebrew (at least until recently, when it switched to `/opt`).

[Technical note: there has been a lot of discussion about whether `/usr/local/bin` is the right place. It is the default location for user-installed software on Unix machines. However, there are arguments against, which is why Homebrew switched to `opt`, starting with "silicon" M1 Macs. One complication of `/opt/` is that it is not on your path so you will have to deal with that. See here, or my previous write-up about path stuff.]

For a first exploration, just type `python3` into Terminal and don't worry about it.

Python 2 is deprecated now, and you should use Python 3. There is one difference you run into all the time, and some others that are more subtle --- we'll get there.

Basic usage

As we saw above, typing just `python3` puts the Terminal (really, the *shell*) into a different mode, called the Python interpreter. You can recognize it by the prompt, which is now `>>>`.

The interpreter is nice for noodling around. For example

```
>>> s = 'abc'
>>> s
'abc'
>>> s[0]
'a'
>>> s[2]
'c'
>>>
```

`s` is the name we give to the variable whose value is the string `'abc'`. The quotes identify what's in between as a string. Either single or double quotes (matched) are OK. `"xyz"` is also a legal string value.

In the interpreter, if you type in the name of a variable the interpreter responds with the actual value of the variable.

Hence `s` alone gives `'abc'`. To pick out an individual letter, you might supply an index. Python indexing starts with `0`.

```
>>> for i in [0,1,2]:
...     print(s[i])
...
a
b
c
>>>
```

The index of the last letter in this 3-letter string is 2. The valid indexes are `0, 1, 2`.

In the code above, `i` is a variable of the type `int` or integer. Only integers are valid as indexes into a string. In this `for` loop, `i` takes on the values `0, 1, 2` successively.

`[0,1,2]` is a Python list. A list is like an array in other languages. The specification of the loop variable `i` ends with a colon `:`. The next line says what to do in the loop.

This could also be done as

```
L = [0, 1, 2]
>>> for i in L:
...     print(s[i])
...
a
b
c
>>>
```

In these examples, it's important that the next line is indented. I didn't type type the dots, I just pressed space four times. This is called Python's whitespace requirement. 4 spaces is standard. You can use a tab, but just don't. In fact some editors will insert spaces when you tab.

The `print` function tells what to print, which appears on the lines below. Each time through the loop, `i` has a different value, and then we pick out the letter `s[i]` and print it. A `print` statment automatically gives a newline after.

One big difference with Python 3 versus the older version is that the print statement must have parentheses around what is to be printed.

The string `s` has a length, obtained with `len`:

```
>>> len(s)
3
>>>
```

The `range` function works like this:

```
>>> for i in range(len(s)):
...     print(s[i])
...
a
b
c
>>>
```

Although for something as simple as a string, `list` would probably be used.

```
>>> list(s)
['a', 'b', 'c']
>>>
```

The interpreter does arithmetic:

```
>>> 1 + 3
4
>>> 2/3
0.6666666666666666
>>> 5 % 2
1
>>>
```

That's another difference between 2 and 3. Nowadays `/` does standard division. If you want integer division use

```
>>> 3//2
1
>>>
```

The fundamental Python data types include `int`s, `float`s and `string`s. Unlike in C, there is no data type for individual characters, they are just one-letter strings.

Another fundamental data type is the boolean values `True` and `False`.

We saw a list above. Lists are arrays. It is an unusual design choice, but in Python, a list may contain values of different types:

```
>>> L = [1, 'a']
>>> L
[1, 'a']
>>>
```

The other important data type is a Python dictionary, which is a kind of hash table. Dictionaries are designed to give extremely rapid lookup. The dictionary construct uses brackets `{}`

```
>>> D = {'a':'apple','b':'banana'}
>>> D
{'a': 'apple', 'b': 'banana'}
>>> D['a']
'apple'
>>>
```

A colon `:` separates key, value pairs. To see all the keys in a dictionary

```
>>> D.keys()
dict_keys(['a', 'b'])
>>>
>>> for k in D:
...     print(D[k])
...
apple
banana
>>>
```

Scripts

The usual way to invoke Python is to provide the path to a script. Let's say I'm in the Desktop directory and I have `script.py` in the same directory. The script contains one line:

```
print('Hello, world!')
```

which gives this

```
> python3 script.py
Hello, world!
>
```

A third way to use Python is to make a script executable. If the first line is

```
#! python3
print('Hello, world!')
```

and the script file has been marked as *executable*

```
> chmod 744 script.py
> ls -al script.py
-rwxr--r--@ 1 telliott  staff  34 May 26 07:40 script.py
>
```

and we remember to pre-pend `./` to the script's name, then from the Terminal I get

```
> ./script.py
Hello, world!
>
```

The `./` is a Unix peculiarity. The `.` means *this* directory and the `/` means in this directory find the file whose name follows.

Reading and writing files

Suppose I am in a directory that contains the file `caesar.txt`:

```
Gallia est omnis divisa in partes tres.
```

In the interpreter (or in a script) I do:

```
>>> fn = 'caesar.txt'
>>> with open(fn) as fh:
...     data = fh.read()
...
>>> data
'Gallia est omnis divisa in partes tres.'
>>>
>>> print(data)
Gallia est omnis divisa in partes tres.
>>>
```

There is a slight difference.

In our code `fn` is the name of the file, `fh` is what I use to refer to a "file handle", which refers to a file that is "open" for reading. "read" or "r" is the default mode. If we wish to write:

```
>>> fiveprimes = [2,3,5,7,11]
>>> ofn = 'primes.txt'
>>> with open(ofn,'w') as fh:
...     fh.write(tenprimes)
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: write() argument must be str, not list
>>>
```

What's wrong is that fiveprimes is a list of integers. We must explicitly convert these to a single string before writing to a file.

```
>>> pL = [str(e) for e in fiveprimes]
>>> with open(ofn,'w') as fh:
...     fh.write('\n'.join(pL))
...
10
>>>
```

The first line is called a *list comprehension*, which is a handy way to do something for every value in a list.

Here, I have used the variable name `e` to refer to the elements of `fiveprimes`, which are the first five primes. Each one is converted to a string by the call `str(e)`, and the results stored in a list called `pL` (my mnemonic for print list).

The `open(ofn,'w')` says to open the file in write mode. (Unix and Python do not care if this name is already used for another file, we'll write right over it).

The last part is `'\n'.join(pL)`. The `join` function makes a single string out of the elements of `pL`, with the first part `'\n'`, called a separator, joining each element. `'\n'` is the Unix new line control character. The resulting file `primes.txt` has newlines:

```
2
3
5
7
11
```

```
>>> s = ' '.join(pL)
>>> with open(ofn,'w') as fh:
...     fh.write(s)
...
10
>>>
```

This gives, instead, for `primes.txt`, a single line

```
2 3 5 7 11
```

That `10` after the `write` is the number of characters written. There is no newline in the file.

The opposite of `join` is `split`.

```
>>> data = 'Gallia est omnis divisa in partes tres.'
>>> L = data.split()
>>>
>>> L
['Gallia', 'est', 'omnis', 'divisa', 'in', 'partes', 'tres.']
>>> [w for w in L if w.endswith('a')]
['Gallia', 'divisa']
>>>
```

Functions

To define a function in Python, use `def`, like this:

```
>>> def f(s):
...     print(s)
...
>>> f('abc')
abc
>>>
```

`f` is the name we've given to this function (the function is a variable in Python). The parentheses are required, as is the colon `:` and also the indentation for the next line.

What's inside `(s)` is the local name for a variable we're providing to the function. In the line `f('abc')` we're "calling" the function and passing in the string `'abc'`, which gets printed.

A real-world example.

The Sieve of Eratosthenes.

```
pL = []

for n in range(2,25):
    is_prime = True
    for p in pL:
        if n % p == 0:
            is_prime = False
    if prime:
        pL.append(n)

print(pL)
```

which gives

```
> p3 script.py
[2, 3, 5, 7, 11, 13, 17, 19, 23]
>
```

Okay. So `pL` starts as an empty list. (We could also say `pL = list()` and have the same result).

Then we have a `for` loop with `n` taking on the integer values in the half-open range `[2,25)`, that is, 2 through 24.

On the first pass, `n = 2` and `pL` is empty, so the line

```
    for p in pL:
```

does nothing, the boolean value `is_prime` remains `True` and `2` is `append` ed or added to pL.

On the second pass, `n = 3` and `pL` is `[2]`, so we check to see `if n % p == 0`, that is, `if 3 % 2 == 0`.

`%` is the "mod" or modulus or remainder operator, it gives what's left if we carry out integer division of 3 by 2. `3 % 2` is equal to 1.

1 is not `==` (equal) to 0, so the result is to skip the next line. Then `3` is added to the list of primes.

The equality `==` and inequality `!=` operators are very common in Python code, and deliberately distinct from `=`, which is the assignment operator.

On the third pass it is the case that `if 4 % 2 == 0:`, so the "flag" `is_prime` is set to `False` for that round

A detail: we got away with calling `print` on a list of integers in this script, whereas the interpreter said that's an error above. I don't know the reason why.

Another: it would be more standard to `continue` or `break` with a loop, but for technical reasons that doesn't work in the above code. An example:

```
>>> already_seen = [2,5]
>>> for i in range(7):
...     if i in already_seen:
...         continue
...     print(i)
...
0
1
3
4
6
>>>
```

Also if you want something to loop and you don't have a convenient loop variable, you might do

```
while True:
    doX()
    if testY():
        break
```

which will `doX()` at least once and then loop until `testY()` returns True.

Imports

Imports give your code access to stuff which isn't provided by default. Probably the most common use is

```
import sys

print(len(sys.argv))
for e in sys.argv:
    print(e)
```

```
> p3 script.py
1
script.py
> p3 script.py a b c
4
script.py
a
b
c
>
```

The `a b c` are called command-line arguments. They give a way to pass in dynamic info to a script.

Another use for imports: I put my super-special code into **utils.py** on the Desktop. This contains

```
def f(s):
    print(s)
```

Now, in **script.py** I put

```
from utils import f

f('Hello, world!')
```

And then from the command line:

```
> p3 script.py
Hello, world!
>
```

There are a lot of useful "modules" that can be imported:

```
>>> from math import pi, e, cos, log
>>> e
2.718281828459045
>>> pi
3.141592653589793
>>>
>>> cos(2 * pi)
1.0
>>> log(e)
1.0
>>>
```

More about lists

A *slice* of a list is another list. We use brackets `[]` like this:

```
>>> L = list('abcdef')
>>> L[:2]
['a', 'b']
>>> L[1:4]
['b', 'c', 'd']
>>> L[-1]
'f'
>>> L[:-1]
['a', 'b', 'c', 'd', 'e']
>>>
```

So the slice `L[i:j]` may be missing one or both values. If so, the first or last+1 value is automatically substituted. `-1` is a convenient way to refer to the last item in a list. Even this works:

```
>>> L[-3:-1]
['d', 'e']
>>>
```

Lists can be sorted easily.

```
>>> L = list('abcde')
>>> L
['a', 'b', 'c', 'd', 'e']
>>> L.reverse()
>>> L
['e', 'd', 'c', 'b', 'a']
>>> L = sorted(L)
>>> L
['a', 'b', 'c', 'd', 'e']
```

```
>>> L = list('abcde')
>>> L = sorted(L, reverse = True)
>>> L
['e', 'd', 'c', 'b', 'a']
>>>
```

Binary data

```
>>> with open('x.txt','w') as fh:
...     fh.write('my data')
...
7
>>>
>>> with open('x.txt','rb') as fh:
...     data = fh.read()
...
>>> data
b'my data'
>>> list(data)
[109, 121, 32, 100, 97, 116, 97]
>>> str(data, encoding = 'utf-8')
'my data'
>>>
```

When we did `open('x.txt','rb')` we said we want to read the file in mode `rb`, so we get binary data.

Python has no idea about a string encoding for this data, and it is best to just think of the data as an array of `int` s in the interval `[0,255]` or maybe as hexadecimal values.

The `str` method can create a string from the data, but it will not assume that it knows the encoding, we must say `utf-8`, which is the default for writing to a file.

```
>>> data
b'my data'
>>> [hex(b) for b in data]
['0x6d', '0x79', '0x20', '0x64', '0x61', '0x74', '0x61']
>>> data.hex()
'6d792064617461'
>>>
```

To interpret it as our text, we must supply an encoding. If you read a textfile in 'r' mode, the encoding *may* be inferred.

It may be of interest to take a look at what was written to disk. Out in the shell we do `hexdump`:

```
> hexdump x.txt
0000000 796d 6420 7461 0061
0000007
>
```

You will note that the bytes are in reverse order. This is what is called little-endian format.

There are 7 characters total in our string, which make 7 bytes in ASCII, and that is legal UTF-8.

The odd number of bytes have been padded at the end with the zero byte, written before the penultimate value because this is little-endian.

We have really have 8 bytes written to disk. `hexdump` is lying to us because it knows the data was really only 7 bytes.

Summary

```
s = 'abc'            # creates the string variable s with value 'abc'
                     # = is the assignment operator

s == 'abc'           # the equality operator, result is True or False
s != 'abc'           # the inequality operator

if s == 'abc':       # an if statement, next line indented
    doX()

int(s)               # convert a string to an int
                     # must be appropriate value
float(s)             # convert a string to a float
```

```
str(2)              # converts the integer 2 to a string
hex(255)            # converts an integer to the hex value, here 0xff

s[0]                # index notation, starting with 0
s[-1]               # the last value in s
len(s)

seen_that = True     # seen_that is a boolean value, True or False

L = s.split(sep)    # splits a string s on any occurrence of sep
                    # sep can be empty "" then splits on "whitespace"

L = [0,1,2]         # creates a list with values [0, 1, 2]
len(L)              # gives the length of a list (or a string)

L = list('abc')     # creates a list with valuees ['a','b','c']
L = [1,'a']         # lists may contain values of different types
sep.join(L)         # works on a list of strings, sep could be '\n'
                    # result is a string joined by sep

L.reverse()         # reverse a list, result value replaces previous
rL = reversed(L)    # returns a reversed list

L.sort()            # sort a list, resulting value replaces previous
sL = sort(L)        # returns a sorted list
sL = sort(L,key=f)  # sort can take a key saying what to sort on
def k(sL):
    return sL[-1]   # an example of a k (a function) that returns
                    # last value and could be used above (key=k)


L = s.split(sep)    # splits a string s on any occurrence of sep
                    # sep can be "" then splits on "whitespace"

D = {'a':'apple',   # creates a dictionary with keys 'a' and 'b'
     'b':'banana'}
for k in D:         # values are accessed through their keys
    print(D[k])

for i in R:         # a loop, where R is a list of values
    doThis()        # a function call like print(s)


3/2                 # standard division, equal to 1.5, a float
```

```
2 * 3               # multiplication, equal to 6
2 ** 3              # exponentiation, equal to 8
3 % 2               # mod or remainder, equal to 1

print(c)            # a built-in function, print

with open(fn) as fh:
   data = fh.read()    # standard file open code, fn is filename

                      # standard file write code, ofn is filename
with open(ofn,'w') as fh:
    fh.write('\n'.join(pL))

with open(fn,'rb') as fh:     # file open to read binary data
   data = fh.read()

def f(s):           # defines function f with argument s
    print(s)        # prints s, usually a string, but exceptions

import sys          # sys is a module
name = sys.argv[0]  # sys.argv are command-line arguments,
                    # script name is first

from math import pi     # import *only* the name pi from math

import math
C = math.pi*d       # all names in math available, but qualified

str(data,
  encoding = 'utf-8')
                    # converts binary data using encoding UTF-8
                    # if data was 0xE2 0x98 0x83 i.e. [226, 152, 131]
                    # or 1110 0010 1001 1000 1000 0011 valid UTF-8
```

example:

```
>>> data = [226,152,131]
>>> bytes(data)
b'\xe2\x98\x83'
>>> str(bytes(data),encoding='utf-8')
'☃'
>>>
```

That's a little snowman. :)