Python Basics

Brian A. Fannin

2020 July 14

# Welcome

Python programming basics: variables, types, lists, dictionaries, functions, dates, strings, dir, help, simulated transactional data, computing Earned Premium

Welcome aboard! In our first session, we will do the following:

|  |  |
| --- | --- |
| What | How long |
| Preliminary stuff: up and running with Colab | 5 min |
| Boring stuff: operations, data types, and package import | 15 min |
| Semi-fun stuff: functions and loops | 15 min |
| Fun stuff: iterable data | 15 min |
| Some other stuff: stats functions, numpy and a tiny bit of matplotlib | 10 min |
| Practical assignment: earned premium | 15 min |

## Preliminary stuff: up and running with Colab

There are several different ways to work with Python. You can run it from the command line, you can use an IDE like Spyder, PyCharm, or even RStudio (!), or you can use a notebook. A notebook requires the least amount of investment up front for instruction and it will cover most of your initial use cases.

A notebook is divided into “cells”. A cell may contain code or text. Text follows the [markdown](https://daringfireball.net/projects/markdown/) format.

The computer perspective:

**REPL**

1. Read
2. Execute
3. Process
4. Loop

The user perspective:

**TWRL**

1. Type
2. Wait
3. Read
4. Loop

Contrast **REPL** / **TWRL** with spreadsheet, SQL, or compiled programming languages (C/C++, Fortran, etc). It’s interactive, imperative (not descriptive) and (more or less) immediate.

Go ahead and type a few basic commands into a cell, type CTRL+ENTER and see what happens.

5 + 6

>>> 11

# Boring stuff: operators, data types and package import

## Objects

An object is an area of storage in memory, similar to a cell in a spreadsheet. You may also see the word “variable” (similar to a random variable in statistics) to describe this concept.

Objects may be created by assigning the result of an operation to the name of the object.

x = 5

Typing the name of a variable will cause the value of that variable to be printed.

x

>>> 5

In a notebook, only the most recent output is printed.

y = 100  
y

>>> 100

x

>>> 5

### Multiple assignment

Python has support for multiple assignment. Check this out:

x = 2  
y = 5  
x, y

>>> (2, 5)

x, y = y, x  
  
x, y

>>> (5, 2)

## Data types

Python data types are about what you would expect from a programming language. Scalar/primitive types are as follows:

* boolean
* integer
* float
* string

There are also complex and bytes. We will not go into those. Just for fun, though, here’s a complex number:

(-4) \*\* (1 / 2)

>>> (1.2246467991473532e-16+2j)

You can query the object type using the type() function.

type(2)

>>> <class 'int'>

type(2.0)

>>> <class 'float'>

type(5 // 2) # We'll see why soon

>>> <class 'int'>

## Casting

You can cast from one to another using functions like:

bool(some\_value) -> boolean int(some\_value) -> integer float(some\_value) -> float str(some\_value) -> string

Upcasting runs as: boolean -> integer -> float -> complex -> string

some\_value = True  
some\_value

>>> True

int(some\_value)

>>> 1

float(some\_value)

>>> 1.0

str(some\_value)

>>> 'True'

Downcasting works when a conversion may be made unambiguously. Note that you may lose precision!

bool('True')

>>> True

int(5.6)

>>> 5

float('5.0')

>>> 5.0

int('5')

>>> 5

x = True  
y = False  
  
x and y

>>> False

x or y

>>> True

type(x)

>>> <class 'bool'>

name = 'Dave'  
  
name

>>> 'Dave'

type(name)

>>> <class 'str'>

x + name

>>> Error in py\_call\_impl(callable, dots$args, dots$keywords): TypeError: unsupported operand type(s) for +: 'bool' and 'str'  
>>>   
>>> Detailed traceback:   
>>> File "<string>", line 1, in <module>

5 \* 'Dave'

>>> 'DaveDaveDaveDaveDave'

### More on strings

Strings are delimited by single or double quotes.

'This is a string'

>>> 'This is a string'

"And so is this"

>>> 'And so is this'

You can create a multi-line string by using triple quotes

"""This string will take more than one line  
Here's the second line.  
"""

>>> Error in py\_call\_impl(callable, dots$args, dots$keywords): SyntaxError: EOF while scanning triple-quoted string literal (<string>, line 1)

To include a single quote character, you can use double quotes, or escape the single quote with a \ character.

print("This'll do for an apostrophe")

>>> This'll do for an apostrophe

print('This\'ll also do')

>>> This'll also do

Speaking of \, if you need one, you’ll need to type it twice.

print("The modulus operator is given by \\")

>>> The modulus operator is given by \

\n and \t are special characters which will give a new line and tab, respectively.

"I'm done with this line.\nTime for another"

>>> "I'm done with this line.\nTime for another"

"Wait for it: \tHere is is"

>>> 'Wait for it: \tHere is is'

## Operators

Most operators are binary. Negation (in a numeric, not a logical sense) is about the only unary operation we’ll encounter. Technically, + is also a unary operator, but it has no effect, so if fairly trivial. It will not convert a negative number to a positive, it simply returns the operand.

-5

>>> -5

+5

>>> 5

+(-5)

>>> -5

### Numeric

Numeric operators are the ones we encounter is basic math. One potential gotcha for R/Excel users is that the caret operator is a bitwise XOR, *not* exponentiation.

2 + 5

>>> 7

2 \* 5

>>> 10

2 / 5

>>> 0.4

2 // 5 # Drop the remainder

>>> 0

2 % 5 # Return the modulus

>>> 2

2 \*\* 5 # Exponentiation

>>> 32

The + and \* operators can also work on character data. We’ll see more of this in a moment.

### Logical comparison

2 == 5 # Note the double equal sign!

>>> False

2 != 5

>>> True

2 < 5

>>> True

2 > 5

>>> False

### Logical operators

Logical operators use the very expressive and brief English words and and or.

True or False # logical OR

>>> True

True and False # logical AND

>>> False

Don’t confuse them with the bitwise operators & and |. They’re equivalent for logical objects, but not so for integers. They don’t work at all for floats.

True | False

>>> True

True or False

>>> True

1 or 4

>>> 1

1 | 4

>>> 5

1.0 or 4.0

>>> 1.0

1.0 | 4.0

>>> Error in py\_call\_impl(callable, dots$args, dots$keywords): TypeError: unsupported operand type(s) for |: 'float' and 'float'  
>>>   
>>> Detailed traceback:   
>>> File "<string>", line 1, in <module>

True & False

>>> False

True and False

>>> False

1 and 4

>>> 4

1 & 4

>>> 0

### Operate and assign

You can use the basic operators with an equal sign to alter an object in place.

x = 5  
x

>>> 5

x += 4  
x

>>> 9

y = "Dave"  
y += " Thomas"  
y

>>> 'Dave Thomas'

### String operations

The + and \* operator can also apply to strings. This has to do with the properties of a list object, which we’ll get to later. Logical comparisons also work.

'Dave' + 'Thomas'

>>> 'DaveThomas'

'Dave' \* 3

>>> 'DaveDaveDave'

'Dave' \* 'Thomas'

>>> Error in py\_call\_impl(callable, dots$args, dots$keywords): TypeError: can't multiply sequence by non-int of type 'str'  
>>>   
>>> Detailed traceback:   
>>> File "<string>", line 1, in <module>

'Dave' < 'Thomas'

>>> True

'Dave' > 'Thomas'

>>> False

float(x)

>>> 9.0

str(x) + name

>>> '9Dave'

### f-strings

An f-string is a way to have objects formatted and placed into a string, as below.

f'It is {x} that my name is {name}.'

>>> 'It is 9 that my name is Dave.'

f'Pi is equal to {3.1415927:.3f}'

>>> 'Pi is equal to 3.142'

name.upper()

>>> 'DAVE'

Note that often methods on objects do not change the objects themselves. Some functions - particularly in Pandas - will modify an object in place.

name

>>> 'Dave'

name = name.upper()  
name

>>> 'DAVE'

## Modules

Functionality beyond what is found in the base install is available by importing other modules. Similar to an Add-in in Excel, or a package in R.

Three ways to do this.

1. from [package] import [item\_1], [item\_2]
2. import [package] as [abbreviation]
3. import [package]

The first option will bring those items into the global namespace.

from datetime import date  
  
christmas = date(2020, 12, 25)

The second option means that you will need to prefix the name of a function with the abbreviation

import datetime as dt  
  
christmas = dt.date(2020, 12, 25)

No abbreviation means that you will to prefix the function name with the full name of the package.

import datetime  
  
christmas = datetime.date(2020, 12, 25)

Python installs with quite a few additional packages. If there is one that you need, you can install via pip or conda at the command line.

pip install scikit-learn or conda install scikit-learn

### Math

You can find some additional mathematical functions and constants in the math package: <https://docs.python.org/3/library/math.html>.

from math import sqrt, pi, e

## Dates

Dates require the datetime package.

from datetime import date, timedelta  
  
christmas = date(2020, 12, 25)  
christmas

>>> datetime.date(2020, 12, 25)

spring\_solstice = date(2020, 4, 31)

>>> Error in py\_call\_impl(callable, dots$args, dots$keywords): ValueError: day is out of range for month  
>>>   
>>> Detailed traceback:   
>>> File "<string>", line 1, in <module>

spring\_solstice

>>> Error in py\_call\_impl(callable, dots$args, dots$keywords): NameError: name 'spring\_solstice' is not defined  
>>>   
>>> Detailed traceback:   
>>> File "<string>", line 1, in <module>

halloween = date(2020, 10, 31)  
  
christmas - halloween

>>> datetime.timedelta(55)

Date arithmetic works like you should expect.

date(2020, 1, 31) - date(2020, 1, 1)

>>> datetime.timedelta(30)

A date represents one instant after midnight.

date(2010, 12, 31) - date(2010, 1, 1)

>>> datetime.timedelta(364)

(christmas - halloween) / (date(2020, 12, 31) - date(2020, 1, 1))

>>> 0.1506849315068493

You can use the replace function to return an object whose elements are changed.

christmas = christmas.replace(year = 1)

## Miscellany

Comments are noted by a single hash mark.

# This is a comment

### Classes/OOP

We won’t cover this. However, these are very good things and you should know about them.

### Coding practices

Code with the same attention to detail that you apply for getting dressed for a job interview. Intent matters. Python has a de facto standard commonly referred to as PEP-8: <https://www.python.org/dev/peps/pep-0008/>.

Quick notes:

* Give your variables sensible, expressive names. loss\_ratio > LR
* Document *why*, not \_what
* Functions should do one and only one thing. Don’t read in a file, construct a plot and save results in one monolithic function
* Code is *read* (by a human!) more often than it is written. Make your code readable.

## Quick exercise:

1. What is the sqaure root of pi?
2. Express your birthday as a date.
3. Use an f-string to wish yourself a happy birthday.
4. Is “A” more or less than “a”? Why?
5. Why does type(5 // 2) return int?

# Semi-fun stuff: functions, loops and packages

Spaces, not braces! The beginning of a code block is marked with a : at the end of the line. Subsequent lines are indented four spaces. An empty line indicates closure of the block. Note that we don’t need to use end if, next, loop, or anything else to indicate that the block is closed.

Blocks are used for:

* Functions
* Conditional blocks, e.g. if/else
* Loops

## Functions

Defined by the keyword def, followed by an argument list within parentheses and then the aforementioned :. The value returned from the function is indicated by return.

def add\_two(num\_one, num\_two):  
 return num\_one + num\_two

Note that we did not specify data types in the function.

add\_two(3, 4)

>>> 7

add\_two('a', 'b')

>>> 'ab'

add\_two(True, False)

>>> 1

add\_two(True, 1)

>>> 2

Functions may return more than one value. This is an amazing and useful thing!

from math import log  
  
def lognormal\_params(mean, sd):  
 cv = sd / mean  
 sigma = sqrt(log(cv \*\* 2 + 1))  
 mu = log(mean) - (sigma \*\* 2 /2)  
 return mu, sigma  
  
mu, sigma = lognormal\_params(10 \*\* 3, 1.5)  
mu, sigma

>>> (6.907754153983403, 0.001499999156281847)

Objects that are local to the function are not preserved

def add\_five(x):  
 the\_result = x + 5  
 return the\_result  
  
add\_five(5)

>>> 10

the\_result

>>> Error in py\_call\_impl(callable, dots$args, dots$keywords): NameError: name 'the\_result' is not defined  
>>>   
>>> Detailed traceback:   
>>> File "<string>", line 1, in <module>

Object names local to the function take precedence. In the block below, the function add\_five() will *not* use the object the\_result which is defined in the global namespace.

the\_result = 100  
add\_five(5)

>>> 10

However, a function *will* use a global variable if it exists.

def add\_const(x):  
 return x + y  
  
y = 4  
add\_const(10)

>>> 14

### Named and default arguments

You can change the order of arguments to a function by specifying them by name. Unlike R, once you’ve specified an argument by name, you can’t rely on position for the other arguments.

def limited\_loss(loss, limit):  
 """  
 This function assumes that losses are from-ground-up  
 """  
 loss = min(loss, limit)  
 return loss  
  
limited\_loss(50000, 10000)

>>> 10000

limited\_loss(limit = 10000, loss = 50000)

>>> 10000

You can also provide sensible defaults to function arguments.

def limited\_loss\_2(loss, limit = 100000, deductible = 5000):  
 """  
 This function assumes that losses are from-ground-up  
 """  
 loss = min(loss, limit)  
 loss = max(0, loss - deductible)  
 return loss  
  
limited\_loss\_2(50000)

>>> 45000

limited\_loss\_2(50000, deductible=250)

>>> 49750

### lambda functions

Python supports the creation of “lambda” functions. These are functions which can be written in one line. They’re used for disposable functions largely for convenience.

cap\_500 = lambda x: min(x, 500)  
cap\_500(250)

>>> 250

cap\_500(1000)

>>> 500

### Quick exercise

1. Create a function that will return the amount of time between a date and the today’s date. You will want to import the today() function from `datetime.

## Conditional execution

x = 100  
if x > 50:  
 print("x is larger than 50")  
else:  
 print(x)

>>> x is larger than 50

Note that control execute at the first True branch.

x = 100  
if x > 50:  
 print("x is larger than 50")  
elif x > 75:  
 print("x is larger than 75")  
else:  
 print(x)

>>> x is larger than 50

## Loops

Python will always iterate over items in a collection. This will be a list, or a tuple, which we’ll talk about shortly. For now, the range() function is a nice one for generating a sequence.

for i in range(5):  
 print(i)

>>> 0  
>>> 1  
>>> 2  
>>> 3  
>>> 4

for i in range(5, 10):  
 print(i)

>>> 5  
>>> 6  
>>> 7  
>>> 8  
>>> 9

for i in range(2, 21, 2):  
 print(i)

>>> 2  
>>> 4  
>>> 6  
>>> 8  
>>> 10  
>>> 12  
>>> 14  
>>> 16  
>>> 18  
>>> 20

Count the number of days in January

from datetime import timedelta  
  
jan\_days = 0  
for day in range(365):  
 test\_day = date.today() + timedelta(days = day)  
 if test\_day.month == 1:  
 jan\_days += 1  
  
jan\_days

>>> 31

## Quick exercises

1. Create a function that will return the amount of time between a date and the today’s date.
2. Create a function that will wish someone a happy birthday, if today happens to be their birthday. If it is not, have it return a message indicating how many days remain until their birthday.

# The fun stuff

Python embraces the concept of iteration over a collection of objects. Contrast this with R or Matlab’s embrace of vectorized data and operations. Similar, but with some important differences.

## Lists

Lists are one of the most common non-trivial data structures. Lists in Python are very similar to lists in R. They represent an ordered set of heterogeneous data.

You can form a list by manually placing items inside square brackets. Items are separated by commas. Trailing commas are not a problem.

my\_list = [0, 1, 2, 3, 4, ]  
my\_list

>>> [0, 1, 2, 3, 4]

Lists may store heterogeneous data types:

my\_het\_list = [0, 1, 2, 'a', 'b', 'c']  
my\_het\_list

>>> [0, 1, 2, 'a', 'b', 'c']

A list may contain another list:

my\_recursive\_list = ['x', 'y', 'z', my\_list]  
my\_recursive\_list

>>> ['x', 'y', 'z', [0, 1, 2, 3, 4]]

Elements in a list may be modified by assigning values.

my\_list[0] = 5  
my\_list

>>> [5, 1, 2, 3, 4]

my\_list[0] = 0  
my\_list

>>> [0, 1, 2, 3, 4]

You may append new elements using the append() function. Note that this will modify the object in place.

my\_list.append(5)  
my\_list

>>> [0, 1, 2, 3, 4, 5]

You can sort a list using sort() and reverse it using reverse(). These will modify the list in place.

my\_list.sort()  
my\_list

>>> [0, 1, 2, 3, 4, 5]

my\_list.reverse()  
my\_list

>>> [5, 4, 3, 2, 1, 0]

my\_list.sort()  
  
sorted(my\_list)

>>> [0, 1, 2, 3, 4, 5]

### Slicing

You can extract individual elements of a list by referring to their ordinal position. Note that indexing is zero-based, in contrast to R, which is one-based. That is, the first element is at position zero. Despite what anyone may tell you, there is no advantage to zero- or one-based indexing.

my\_list[0]

>>> 0

my\_list[1]

>>> 1

my\_list[4]

>>> 4

Passing in a negative number will extract elements starting from the end.

my\_list[-1]

>>> 5

my\_list[-3]

>>> 3

Multiple elements may be extracted using the : slice operator.

[start]:[end]:[step]

Note that [end] is not inclusive.

Omission of one of the elements means:

* If step is blank, it will be one
* If the start is beginning, it will begin at the first element
* If the end is missing, it will end with the last element

Negative:

* Negative start means that it will begin from the end of the string
* Negative end
* Negative step means that results will be reversed

my\_list[0:4]

>>> [0, 1, 2, 3]

my\_list[0:4:2]

>>> [0, 2]

my\_list[:3]

>>> [0, 1, 2]

my\_list[::-1]

>>> [5, 4, 3, 2, 1, 0]

my\_list[-3:]

>>> [3, 4, 5]

You can use slicing for assignment of multiple items

my\_list[:2] = [11, 12]  
my\_list

>>> [11, 12, 2, 3, 4, 5]

The magic of multiple assignment means we can extract to more than one variable at a time

x, y = my\_list[:2]  
x, y

>>> (11, 12)

Fun fact: a string is actually a list!

my\_name = 'Brian'  
my\_name[0]

>>> 'B'

my\_name[-1]

>>> 'n'

Revisit the + and \* operators on strings which we saw earlier. These are general to lists.

my\_list \* 2

>>> [11, 12, 2, 3, 4, 5, 11, 12, 2, 3, 4, 5]

my\_list + [1,2]

>>> [11, 12, 2, 3, 4, 5, 1, 2]

Quick exercise:

1. Pull out every third letter from your name
2. Extract the penultimate letter of your name

## Tuples

A tuple is very similar to a list. The most significant difference is that the data is *immutable*. This means that items inside the tuple may not be changed. They are created similar to lists, but use parentheses rather than square brackets.

my\_tuple = (3, 5)  
my\_tuple[0]

>>> 3

my\_tuple[0] = 4

>>> Error in py\_call\_impl(callable, dots$args, dots$keywords): TypeError: 'tuple' object does not support item assignment  
>>>   
>>> Detailed traceback:   
>>> File "<string>", line 1, in <module>

Tuples exist, but you will not often create them. They are often returned from functions, so you will see them.

## Sets

A set is a collection of unique items. Like a tuple the elements are not mutable.

set('Brian Fannin')

>>> {'B', ' ', 'r', 'i', 'F', 'n', 'a'}

set([1, 2, 2, 3, 3, 3, 4, 4, 4, 4])

>>> {1, 2, 3, 4}

## Dictionaries

A dictionary is a set of key-value pairs. The values may be lists. This is similar to JSON/XML. It may also remind you of a structure in C, or user-defined type in VBA.

A dictionary may be created by placing key-value pairs in parentheses, or calling the dict() function.

my\_dict = {  
 'effective\_date': date(2020, 7, 1),  
 'expiration\_date': date(2021, 6, 30),  
 'premium': 10e3,  
}  
  
my\_dict

>>> {'effective\_date': datetime.date(2020, 7, 1), 'expiration\_date': datetime.date(2021, 6, 30), 'premium': 10000.0}

my\_other\_dict = dict(  
 effective\_date = date(2020, 7, 1),  
 expiration\_date = date(2021, 6, 30),  
 premium = 10e3,  
)  
my\_other\_dict

>>> {'effective\_date': datetime.date(2020, 7, 1), 'expiration\_date': datetime.date(2021, 6, 30), 'premium': 10000.0}

my\_dict == my\_other\_dict

>>> True

Elements may be accessed by name, but not position.

my\_dict['effective\_date']

>>> datetime.date(2020, 7, 1)

my\_dict[0]

>>> Error in py\_call\_impl(callable, dots$args, dots$keywords): KeyError: 0  
>>>   
>>> Detailed traceback:   
>>> File "<string>", line 1, in <module>

Reassignment doesn’t care about data types

my\_dict['effective\_date'] = 7  
my\_dict['effective\_date']

>>> 7

New keys may be added simply through assignment

my\_dict['mojo'] = 'cazart'  
my\_dict

>>> {'effective\_date': 7, 'expiration\_date': datetime.date(2021, 6, 30), 'premium': 10000.0, 'mojo': 'cazart'}

You may investigate what keys are included by calling the keys() method.

my\_dict.keys()

>>> dict\_keys(['effective\_date', 'expiration\_date', 'premium', 'mojo'])

values() will give you the, um, values.

my\_dict.values()

>>> dict\_values([7, datetime.date(2021, 6, 30), 10000.0, 'cazart'])

You can see which keys exist in a dictionary by calling in.

'effective\_date' in my\_dict

>>> True

'line\_of\_business' in my\_dict

>>> False

### Why dictionaries?

A tangible benefit of dictionaries is that they are fast. The time to access an item in a dictionary is independent of the size of the dictionary. This comes from some cost to add items, particularly if the order of items is important and you want to add something to the middle of the dictionary.

The less tangible benefit is that the elements are expressive.

## Comprehensions

Lists and dictionaries are *iterable*. We can walk through the elements in sequence and perform operations to create new lists and dictionaries. This is closely associated with list comprehensions and dictionary comprehensions.

### List comprehension

A list comprehension (sometimes abbreviated “listcomp”) will generate a list. This is more compact way of achieving something for which you may have used a for loop.

[expression for list\_element in a\_list] [expression for list\_element in a\_list if *condition*]

[add\_two(x, 1) for x in range(2)]

>>> [1, 2]

Get all the days in February 2020:

february\_days = [date(2020, 2, x) for x in range(1, 30)]

We can use multiple assignment to create the same effect as a nested loop

[add\_two(x, y) for x in range(2)   
 for y in range(2)]

>>> [0, 1, 1, 2]

my\_last\_name = 'Fannin'  
  
[add\_two(x, y) for x in my\_name   
 for y in my\_last\_name]

>>> ['BF', 'Ba', 'Bn', 'Bn', 'Bi', 'Bn', 'rF', 'ra', 'rn', 'rn', 'ri', 'rn', 'iF', 'ia', 'in', 'in', 'ii', 'in', 'aF', 'aa', 'an', 'an', 'ai', 'an', 'nF', 'na', 'nn', 'nn', 'ni', 'nn']

A list comprehension will also let us create a logical subset of a list. We can check membership in a list by using the keyword in.

[x for x in my\_name if x in 'aeiou']

>>> ['i', 'a']

Quick question

1. How could you use list comprehension to create a list of quarter start dates for the years 2001 through 2005?

### Dictionary comprehension

Similar to a list comprehension. Note that we can peform an operation on the value which will form the key.

new\_dict = {str(x): x\*\*2 for x in [1, 2, 3, 4]}  
new\_dict

>>> {'1': 1, '2': 4, '3': 9, '4': 16}

new\_dict.keys()

>>> dict\_keys(['1', '2', '3', '4'])

new\_dict.values()

>>> dict\_values([1, 4, 9, 16])

new\_dict['1']

>>> 1

We can do this with multiple elements, to create a dictionary of lists

{  
 'written\_premium': [100, 200]  
 , 'effective\_date': [date(a\_year, 1, 1) for a\_year in range(2001, 2003)]  
}

>>> {'written\_premium': [100, 200], 'effective\_date': [datetime.date(2001, 1, 1), datetime.date(2002, 1, 1)]}

## Iteration

Python supports many iterable expressions. These can be thought of as functions which generate the next value in a sequence.

### Generator expression

Generator expressions look similar to a list comprehension, but are enclosed within parentheses rather than square brackets.

We’ll create a simple function which determines if an integer is odd.

def is\_odd(x):  
 return x % 2 != 0  
  
is\_odd(8)

>>> False

We can use list comprehension to form a Boolean list of the oddness of numbers.

mojo = [is\_odd(x) for x in range(10)]  
mojo

>>> [False, True, False, True, False, True, False, True, False, True]

A generator function will not run execute until you ask it to.

gonzo = (is\_odd(x) for x in range(10))  
gonzo

>>> <generator object <genexpr> at 0x000000002B4C49E8>

So, let’s ask it.

cazart = list(gonzo)  
cazart

>>> [False, True, False, True, False, True, False, True, False, True]

Note that the two lists are identical

cazart == mojo

>>> True

The difference:

* A list comprehension is *greedy*. It will produce all results at once.
* A generator expression is *lazy*. It will generate the next result in a sequence as needed

We can use next() to ask for the next item in the sequence:

gonzo = (is\_odd(x) for x in range(10))  
next(gonzo)

>>> False

next(gonzo)

>>> True

### zip()

zip() will pair up matching items in two or more lists.

my\_zip = zip(  
 ['a', 'b', 'c'],  
 [1, 2, 3]  
)  
my\_zip

>>> <zip object at 0x000000002B6DA288>

In Python 3, zip() will return an iterator. We can extract the results using list().

zipped\_stuff = list(my\_zip)  
zipped\_stuff

>>> [('a', 1), ('b', 2), ('c', 3)]

for item in my\_zip:  
 print(item)

ids = [str(i) for i in range(100)]  
written\_premium = [float(i) / 2 for i in range(100)]  
  
policies = dict(zip(ids, written\_premium))  
policies['3']

>>> 1.5

### enumerate()

enumerate() will produce each item in a list, along with the index of the item.

my\_list = list('Brian')  
for i, letter in enumerate(my\_list):  
 f'Item {i} is {letter}'

>>> 'Item 0 is B'  
>>> 'Item 1 is r'  
>>> 'Item 2 is i'  
>>> 'Item 3 is a'  
>>> 'Item 4 is n'

We can use enumerate() with the generator we created earlier.

gonzo = (is\_odd(x) for x in range(10))  
for i, oddness in enumerate(gonzo):  
 f'The number {i} is odd: {oddness}'

>>> 'The number 0 is odd: False'  
>>> 'The number 1 is odd: True'  
>>> 'The number 2 is odd: False'  
>>> 'The number 3 is odd: True'  
>>> 'The number 4 is odd: False'  
>>> 'The number 5 is odd: True'  
>>> 'The number 6 is odd: False'  
>>> 'The number 7 is odd: True'  
>>> 'The number 8 is odd: False'  
>>> 'The number 9 is odd: True'

### yield

You can construct a generator function by using the yield keyword. This will look a lot like a function. However, instead of using return, you will use yield. This will return the next value in a sequence. Below we create a generator which will count integers beginning at zero.

def counter(limit):  
 the\_counter = 0  
 while the\_counter < limit:  
 yield the\_counter  
 the\_counter += 1  
  
count\_four = counter(4)  
  
next(count\_four)

>>> 0

next(count\_four)

>>> 1

next(count\_four)

>>> 2

next(count\_four)

>>> 3

next(count\_four)

>>> Error in py\_call\_impl(callable, dots$args, dots$keywords): StopIteration:   
>>>   
>>> Detailed traceback:   
>>> File "<string>", line 1, in <module>

We can use list comprehension to create a list using a generator.

new\_counter = counter(10)  
my\_list = [next(new\_counter) for \_ in range(10)]  
my\_list

>>> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Other stuff

## scipy.stats

<https://docs.scipy.org/doc/scipy/reference/stats.html>

scipy stats has some useful stats functions, including random number generators, cumulative and density functions, etc.

from scipy.stats import poisson, describe  
  
sims = 10 \*\* 3  
count\_mean = 5  
arr\_pois = poisson.rvs(count\_mean, size = sims)  
  
sum(arr\_pois)

>>> 4958

describe(arr\_pois)

>>> DescribeResult(nobs=1000, minmax=(0, 13), mean=4.958, variance=4.935171171171172, skewness=0.4491834159528475, kurtosis=0.02156716212365417)

Notice that I didn’t create an object called lambda. lambda is a reserved word in Python.

## numpy

<https://numpy.org/doc/stable/index.html>

In our next session, we will begin to use pandas. This will remain our preferred means of handling data sets. pandas uses numpy under the hood. Benefits of using numpy:

* Vectorized operations
* Smaller memory footprint
* Faster execution

import numpy as np  
  
an\_array = np.arange(5)  
an\_array

>>> array([0, 1, 2, 3, 4])

an\_array \* 2

>>> array([0, 2, 4, 6, 8])

### numpy stats

numpy replicates quite a few of the stats functions from scipy. However, they’re optimized for array-like objects.

arr\_pois\_2 = np.random.poisson(count\_mean, sims)  
describe(arr\_pois\_2)

>>> DescribeResult(nobs=1000, minmax=(0, 15), mean=5.032, variance=5.322298298298299, skewness=0.42931955882946793, kurtosis=0.2741009272010948)

## matplotlib

We’ll get into visualization later, but can’t let the first session pass without a quick mention. The code below will let you get a basic scatter plot (which handles 80% of your use cases) and a histogram (which handles another 10%).

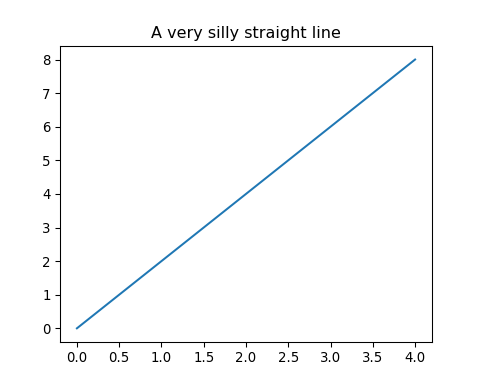
Within a Jupyter notebook, you won’t need to call plt.show().

To use matplotlib within Jupyter, we’ll need to call the “magic” command %matplotlib inline.

%matplotlib inline

import matplotlib.pyplot as plt

fig, axis = plt.subplots()  
  
axis.plot(an\_array, an\_array \* 2)  
axis.set\_title("A very silly straight line")

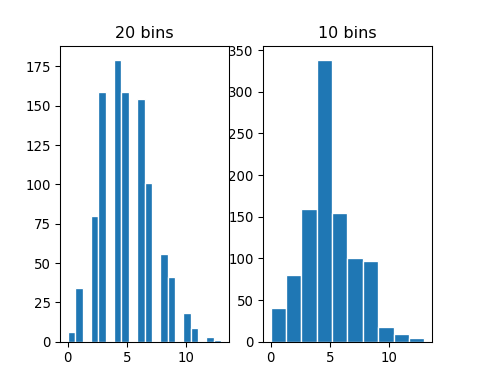


We can also have more than one plot

fig, (axis1, axis2) = plt.subplots(1, 2)  
  
axis1.hist(arr\_pois, bins = 20, ec = 'w')

axis1.set\_title("20 bins")  
axis2.hist(arr\_pois, bins = 10, ec = 'w')

axis2.set\_title("10 bins")



matplotlib is utilitarian, like R’s base graphics. It’s OK for getting started, but we’ll show some better options.

# A practical assignment

We would like to randomly generate a set of policies. Nothing fancy. Each policy will contain an effective and expiration date and written premium. Next, we’d like to calculate the earned premium by calendar year.

## Random set of policyholders

We’ll start by importing some functions that we’ll need.

from datetime import date, timedelta  
from random import seed, randrange

Next, we’ll create a lambda function to increment the effective date by a year. This is largely to make the code more readable.

increment\_year = lambda date\_in :date(date\_in.year + 1, date\_in.month, date\_in.day)

Finally, we’ll create a generator which gives us a randomized policy. There’s not much special about the written premium. It’s simply a number between 0 and one thousand.

Note 1) the use of our lambda function, 2) the use of a random number seed and 3) the yield keyword which makes this a generator.

def random\_policy(new\_seed = 1234):  
 """  
 This is a generator which produces a dictionary of basic policy information  
 """  
 seed(new\_seed)  
 while True:  
 policy = {  
 'written\_premium': randrange(10\*\*3),  
 'effective\_date': date(2020, 1, 1) + timedelta(randrange(365))  
 }  
 policy['expiration\_date'] = increment\_year(policy['effective\_date'])  
 yield policy

We can now generate a list of dictionaries by calling next() as many times as we like. We don’t need the result of our call to range(). We indicate that we’re ignoring it by assigning it to the \_ object.

policy\_generator = random\_policy()  
  
policies = [next(policy\_generator) for \_ in range(10)]

Take a look at our policies:

policies[:2]

>>> [{'written\_premium': 989, 'effective\_date': datetime.date(2020, 8, 13), 'expiration\_date': datetime.date(2021, 8, 13)}, {'written\_premium': 119, 'effective\_date': datetime.date(2020, 1, 4), 'expiration\_date': datetime.date(2021, 1, 4)}]

And confirm that we have as many as we’re expecting:

len(policies)

>>> 10

## Earned premium

Compose a function to give the earned premium based on two dates. For example, a policy written on July 1, 2020 is roughly 50% earned between July 1, 2020 and December 31, 2020.

def earned\_premium(written\_premium, policy\_start, policy\_end, earn\_start, earn\_end):  
 """  
 Calculate earned premium. Presumes an even (linear) earning pattern  
 """  
 policy\_term = policy\_end - policy\_start  
 earn\_start = max(earn\_start, policy\_start)  
 earn\_end = min(earn\_end, policy\_end)  
 frac\_earned = (earn\_end - earn\_start) / (policy\_end - policy\_start)  
   
 return written\_premium \* frac\_earned

We can confirm that the function works with some sample data.

premium = 10\*\*3  
earned\_premium(premium, date(2020, 7, 1), date(2021, 7, 1), date(2020, 1, 1), date(2021, 1, 1))

>>> 504.10958904109594

earned\_premium(premium, date(2020, 7, 1), date(2021, 7, 1), date(2021, 1, 1), date(2022, 1, 1))

>>> 495.8904109589041

This function will return the calendar year intervals which bracket a list of dates. Note the use of set().

def get\_cy\_intervals(dates\_list):  
 """  
 This function will return a list of calendar years  
 """  
 the\_years = set([a\_date.year for a\_date in dates\_list])  
 cy\_year = [[date(a\_year, 1, 1), date(a\_year + 1, 1, 1)] for a\_year in the\_years]  
 return cy\_year

We can see it in action here:

some\_dates = [  
 date(2020, 7, 1)  
 , date(2020, 12, 31)  
 , date(2021, 6, 30)  
]  
  
get\_cy\_intervals(some\_dates)

>>> [[datetime.date(2020, 1, 1), datetime.date(2021, 1, 1)], [datetime.date(2021, 1, 1), datetime.date(2022, 1, 1)]]

Those two functions will let us compute the earned premium in all of the calendar years in a policy coverage period.

def earned\_premium\_cy(written\_premium, policy\_start, policy\_end):  
 """  
 Calculate the amount of premium earned by calendar year for a single policy period  
 """  
 cy\_return = {}  
 for cy\_start, cy\_end in get\_cy\_intervals([policy\_start, policy\_end]):  
 cy\_return[cy\_start.year] = earned\_premium(  
 written\_premium, policy\_start, policy\_end, cy\_start, cy\_end  
 )  
 return cy\_return

Does it work?

earned\_premium\_cy(10e3, date(2020, 7, 1), date(2021, 6, 30))

>>> {2020: 5054.945054945055, 2021: 4945.054945054945}

It does!

## All together now

# Initialize empty dictionary to store total earned premium by calendar year  
cy\_earned = {}  
for policy in policies:  
   
 cy\_ep = earned\_premium\_cy(policy['written\_premium'], policy['effective\_date'], policy['expiration\_date'])  
   
 for year in cy\_ep:  
 if year in cy\_earned:  
 cy\_earned[year] += cy\_ep[year]  
 else:  
 cy\_earned[year] = cy\_ep[year]  
  
cy\_earned

>>> {2020: 1706.729598023804, 2021: 1602.270401976196}

Here’s a sneak preview of what’s happening next week:

import pandas as pd  
  
pd.DataFrame(policies)

>>> written\_premium effective\_date expiration\_date  
>>> 0 989 2020-08-13 2021-08-13  
>>> 1 119 2020-01-04 2021-01-04  
>>> 2 92 2020-10-25 2021-10-25  
>>> 3 35 2020-12-09 2021-12-09  
>>> 4 709 2020-02-12 2021-02-12  
>>> 5 100 2020-06-30 2021-06-30  
>>> 6 242 2020-01-09 2021-01-09  
>>> 7 31 2020-01-09 2021-01-09  
>>> 8 354 2020-11-27 2021-11-27  
>>> 9 638 2020-09-04 2021-09-04

For the student

1. Simulate 5,000 simulations from a Poisson with an expected value of 5
2. Do the same for a negative binomial. Use values of 0.2, 0.5 and 1.0 for the coefficient of variation