Quantitative Economics with Python

Thomas J. Sargent John Stachurski

Update: New build system

QuantEcon is migrating to a new build system - please report any errors to contact@quantecon.org

How to read this lecture...

Python Essentials

In this lecture we'll cover features of the language that are essential to reading and writing Python code

Data Types

We've already met several built in Python data types, such as strings, integers, floats and lists

Let's learn a bit more about them

Primitive Data Types

One simple data type is **Boolean values**, which can be either `True` or `False`

In
```
x = True
x
```

Out
```
True
```

In the next line of code, the interpreter evaluates the expression on the right of = and binds y to this value

In
```
y = 100 < 10
y
```

Out
```
False
```

In
```
type(y)
```
Out
```
bool
```

In arithmetic expressions, `True` is converted to `1` and `False` is converted `0`

This is called **Boolean arithmetic** and is often useful in programming

Here are some examples

In
```
x + y
```
Out
```
1
```
In
```
x * y
```
Out
```
0
```
In
```
True + True
```
Out
```
2
```
In
```
bools = [True, True, False, True]  # List of Boolean values

sum(bools)
```
Out
```
3
```

The two most common data types used to represent numbers are integers and floats

In
```
a, b = 1, 2
c, d = 2.5, 10.0
type(a)
```
Out
```
int
```
In
```
type(c)
```
Out
```
float
```

Computers distinguish between the two because, while floats are more informative, arithmetic operations on integers are faster and more accurate

As long as you're using Python 3.x, division of integers yields floats

In
```
1 / 2
```
Out
```
0.5
```

But be careful! If you're still using Python 2.x, division of two integers returns only the integer part

For integer division in Python 3.x use this syntax:

In
```
1 // 2
```
Out
```
0
```

Complex numbers are another primitive data type in Python

In
```
x = complex(1, 2)
y = complex(2, 1)
x * y
```

Out
```
5j
```

Containers

Python has several basic types for storing collections of (possibly heterogeneous) data

We've already discussed lists

A related data type is **tuples**, which are "immutable" lists

In
```
x = ('a', 'b')  # Parentheses instead of the square brackets
x = 'a', 'b'    # Or no brackets --- the meaning is identical
x
```

Out
```
('a', 'b')
```

In
```
type(x)
```

Out
```
tuple
```

In Python, an object is called **immutable** if, once created, the object cannot be changed

Conversely, an object is **mutable** if it can still be altered after creation

Python lists are mutable

In
```
x = [1, 2]
x[0] = 10
x
```

Out
```
[10, 2]
```

But tuples are not

In
```
x = (1, 2)
x[0] = 10
```

Out
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-16-d1b2647f6c81> in <module>()
      1 x = (1, 2)
----> 2 x[0] = 10

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

We'll say more about the role of mutable and immutable data a bit later

Tuples (and lists) can be "unpacked" as follows

In
```
integers = (10, 20, 30)
x, y, z = integers
x
```

Out
```
10
```

In
```
y
```

Out
```
20
```

You've actually seen an example of this already

Tuple unpacking is convenient and we'll use it often

Slice Notation

To access multiple elements of a list or tuple, you can use Python's slice notation

For example,

In
```
a = [2, 4, 6, 8]
a[1:]
```

Out
```
[4, 6, 8]
```

In
```
a[1:3]
```

Out
```
[4, 6]
```

The general rule is that `a[m:n]` returns `n - m` elements, starting at `a[m]`

Negative numbers are also permissible

In
```
a[-2:]  # Last two elements of the list
```

Out
```
[6, 8]
```

The same slice notation works on tuples and strings

In
```
s = 'foobar'
s[-3:]  # Select the last three elements
```

Out
```
'bar'
```

Sets and Dictionaries

Two other container types we should mention before moving on are sets and dictionaries

Dictionaries are much like lists, except that the items are named instead of numbered

In
```
d = {'name': 'Frodo', 'age': 33}
type(d)
```

Out
```
dict
```

In
```
d['age']
```

Out
```
33
```

The names `'name'` and `'age'` are called the keys

The objects that the keys are mapped to (`'Frodo'` and `33`) are called the `values`

Sets are unordered collections without duplicates, and set methods provide the usual set theoretic operations

In
```
s1 = {'a', 'b'}
type(s1)
```

Out
```
set
```

In
```
s2 = {'b', 'c'}
s1.issubset(s2)
```

Out
```
False
```

In
```
s1.intersection(s2)
```

Out
```
{'b'}
```

The `set()` function creates sets from sequences

In
```
s3 = set(('foo', 'bar', 'foo'))
s3
```

Out
```
{'bar', 'foo'}
```

Input and Output

Let's briefly review reading and writing to text files, starting with writing

In
```
f = open('newfile.txt', 'w')   # Open 'newfile.txt' for writing
f.write('Testing\n')           # Here '\n' means new line
f.write('Testing again')
f.close()
```

Here

- The built-in function `open()` creates a file object for writing to
- Both `write()` and `close()` are methods of file objects

Where is this file that we've created?

Recall that Python maintains a concept of the present working directory (pwd) that can be located from with Jupyter or IPython via

In
```
%pwd
```

Out
```
'/home/quantecon/repos-collab/quantecon.build.lectures/_build_jupyter/py'
```

If a path is not specified, then this is where Python writes to

We can also use Python to read the contents of `newline.txt` as follows

In
```
f = open('newfile.txt', 'r')
out = f.read()
out
```

Out
```
'Testing\nTesting again'
```

In
```
print(out)
```

Out
```
Testing
Testing again
```

Paths

Note that if `newfile.txt` is not in the present working directory then this call to `open()` fails

In this case you can shift the file to the pwd or specify the full path to the file

```
f = open('insert_full_path_to_file/newfile.txt', 'r')
```

Iterating

One of the most important tasks in computing is stepping through a sequence of data and performing a given action

One of Python's strengths is its simple, flexible interface to this kind of iteration via the `for` loop

Looping over Different Objects

Many Python objects are “iterable”, in the sense that they can looped over

To give an example, let’s write the file us_cities.txt, which lists US cities and their population, to the present working directory

In
```
%%file us_cities.txt
new york: 8244910
los angeles: 3819702
chicago: 2707120
houston: 2145146
philadelphia: 1536471
phoenix: 1469471
san antonio: 1359758
san diego: 1326179
dallas: 1223229
```

Out
```
Writing us_cities.txt
```

Suppose that we want to make the information more readable, by capitalizing names and adding commas to mark thousands

The program us_cities.py program reads the data in and makes the conversion:

In
```
data_file = open('us_cities.txt', 'r')
for line in data_file:
    city, population = line.split(':')         # Tuple unpacking
    city = city.title()                        # Capitalize city names
    population = f'{int(population):,}'        # Add commas to numbers
    print(city.ljust(15) + population)
data_file.close()
```

Out
```
New York       8,244,910
Los Angeles    3,819,702
Chicago        2,707,120
Houston        2,145,146
Philadelphia   1,536,471
Phoenix        1,469,471
San Antonio    1,359,758
San Diego      1,326,179
Dallas         1,223,229
```

Here `format()` is a string method used for inserting variables into strings

The reformatting of each line is the result of three different string methods, the details of which can be left till later

The interesting part of this program for us is line 2, which shows that

1. The file object `f` is iterable, in the sense that it can be placed to the right of `in` within a `for` loop
2. Iteration steps through each line in the file

This leads to the clean, convenient syntax shown in our program

Many other kinds of objects are iterable, and we’ll discuss some of them later on

Looping without Indices

One thing you might have noticed is that Python tends to favor looping without explicit indexing

For example,

In
```
x_values = [1, 2, 3]  # Some iterable x
for x in x_values:
    print(x * x)
```

Out
```
1
4
9
```

is preferred to

In
```
for i in range(len(x_values)):
    print(x_values[i] * x_values[i])
```

Out
```
1
4
9
```

When you compare these two alternatives, you can see why the first one is preferred

Python provides some facilities to simplify looping without indices

One is `zip()`, which is used for stepping through pairs from two sequences

For example, try running the following code

In
```
countries = ('Japan', 'Korea', 'China')
cities = ('Tokyo', 'Seoul', 'Beijing')
for country, city in zip(countries, cities):
    print(f'The capital of {country} is {city}')
```

Out
```
The capital of Japan is Tokyo
The capital of Korea is Seoul
The capital of China is Beijing
```

The `zip()` function is also useful for creating dictionaries — for example

In
```
names = ['Tom', 'John']
marks = ['E', 'F']
dict(zip(names, marks))
```

Out
```
{'Tom': 'E', 'John': 'F'}
```

If we actually need the index from a list, one option is to use `enumerate()`

To understand what `enumerate()` does, consider the following example

In
```
letter_list = ['a', 'b', 'c']
for index, letter in enumerate(letter_list):
    print(f"letter_list[{index}] = '{letter}'")
```

Out
```
letter_list[0] = 'a'
letter_list[1] = 'b'
letter_list[2] = 'c'
```

The output of the loop is

In
```
letter_list[0] = 'a'
letter_list[1] = 'b'
letter_list[2] = 'c'
```

Comparisons and Logical Operators

Comparisons

Many different kinds of expressions evaluate to one of the Boolean values (i.e., `True` or `False`)

A common type is comparisons, such as

In
```
x, y = 1, 2
x < y
```

Out
```
True
```

In
```
x > y
```

Out
```
False
```

One of the nice features of Python is that we can chain inequalities

In
```
1 < 2 < 3
```

Out
```
True
```

In
```
1 <= 2 <= 3
```

Out
```
True
```

As we saw earlier, when testing for equality we use `==`

In
```
x = 1    # Assignment
x == 2   # Comparison
```

Out
```
False
```

For “not equal” use `!=`

In
```
1 != 2
```

Out
```
True
```

Note that when testing conditions, we can use **any** valid Python expression

In
```
x = 'yes' if 42 else 'no'
x
```

Out
```
'yes'
```

In
```
x = 'yes' if [] else 'no'
x
```

Out
```
'no'
```

What’s going on here?

The rule is:

- Expressions that evaluate to zero, empty sequences or containers (strings, lists, etc.) and `None` are all equivalent to `False`
    - for example, `[]` and `()` are equivalent to `False` in an `if` clause
- All other values are equivalent to `True`
    - for example, `42` is equivalent to `True` in an `if` clause

Combining Expressions

We can combine expressions using `and`, `or` and `not`

These are the standard logical connectives (conjunction, disjunction and denial)

In
```
1 < 2 and 'f' in 'foo'
```

Out
```
True
```

In
```
1 < 2 and 'g' in 'foo'
```

Out
```
False
```

In
```
1 < 2 or 'g' in 'foo'
```

Out
```
True
```

In
```
not True
```

Out
```
False
```

In
```
not not True
```

Out
```
True
```

Remember

- `P and Q` is `True` if both are `True`, else `False`
- `P or Q` is `False` if both are `False`, else `True`

More Functions

Let's talk a bit more about functions, which are all-important for good programming style

Python has a number of built-in functions that are available without `import`

We have already met some

In
```
max(19, 20)
```

Out
```
20
```

In
```
range(4)  # in python3 this returns a range iterator object
```

Out
```
range(0, 4)
```

In
```
list(range(4))  # will evaluate the range iterator and create a list
```

Out
```
[0, 1, 2, 3]
```

In
```
str(22)
```

Out
```
'22'
```

In
```
type(22)
```

Out
```
int
```

Two more useful built-in functions are `any()` and `all()`

In
```
bools = False, True, True
all(bools)  # True if all are True and False otherwise
```

Out
```
False
```

In
```
any(bools)  # False if all are False and True otherwise
```

Out
```
True
```

The full list of Python built-ins is here

Now let's talk some more about user-defined functions constructed using the keyword `def`

Why Write Functions?

User defined functions are important for improving the clarity of your code by

- separating different strands of logic
- facilitating code reuse

(Writing the same thing twice is almost always a bad idea)

The basics of user defined functions were discussed here

The Flexibility of Python Functions

As we discussed in the previous lecture, Python functions are very flexible

In particular

- Any number of functions can be defined in a given file
- Functions can be (and often are) defined inside other functions
- Any object can be passed to a function as an argument, including other functions
- A function can return any kind of object, including functions

We already gave an example of how straightforward it is to pass a function to a function

Note that a function can have arbitrarily many `return` statements (including zero)

Execution of the function terminates when the first return is hit, allowing code like the following example

In
```
def f(x):
    if x < 0:
        return 'negative'
    return 'nonnegative'
```

Functions without a return statement automatically return the special Python object `None`

Docstrings

Python has a system for adding comments to functions, modules, etc. called docstrings

The nice thing about docstrings is that they are available at run-time

Try running this

In
```
def f(x):
    """
    This function squares its argument
    """
    return x**2
```

After running this code, the docstring is available

In
```
f?
```

```
Type:       function
String Form:<function f at 0x2223320>
File:       /home/john/temp/temp.py
Definition: f(x)
Docstring:  This function squares its argument
```

In
```
f??
```

```
Type:       function
String Form:<function f at 0x2223320>
File:       /home/john/temp/temp.py
Definition: f(x)
Source:
def f(x):
    """
    This function squares its argument
    """
    return x**2
```

With one question mark we bring up the docstring, and with two we get the source code as well

One-Line Functions: `lambda`

The `lambda` keyword is used to create simple functions on one line

For example, the definitions

In
```
def f(x):
    return x**3
```

and

In
```
f = lambda x: x**3
```

are entirely equivalent

To see why `lambda` is useful, suppose that we want to calculate $\int_0^2 x^3 dx$ (and have forgotten our high-school calculus)

The SciPy library has a function called `quad` that will do this calculation for us

The syntax of the `quad` function is `quad(f, a, b)` where `f` is a function and `a` and `b` are numbers

To create the function $f(x) = x^3$ we can use `lambda` as follows

In
```
from scipy.integrate import quad

quad(lambda x: x**3, 0, 2)
```

lut
```
(4.0, 4.440892098500626e-14)
```

Here the function created by `lambda` is said to be anonymous, because it was never given a name

Keyword Arguments

If you did the exercises in the previous lecture, you would have come across the statement

```
plt.plot(x, 'b-', label="white noise")
```

In this call to Matplotlib's `plot` function, notice that the last argument is passed in `name=argument` syntax

This is called a keyword argument, with `label` being the keyword

Non-keyword arguments are called positional arguments, since their meaning is determined by order

- `plot(x, 'b-', label="white noise")` is different from `plot('b-', x, label="white noise")`

Keyword arguments are particularly useful when a function has a lot of arguments, in which case it's hard to remember the right order

You can adopt keyword arguments in user defined functions with no difficulty

The next example illustrates the syntax

In
```
def f(x, a=1, b=1):
    return a + b * x
```

The keyword argument values we supplied in the definition of `f` become the default values

In
```
f(2)
```

Out
```
3
```

They can by modified as follows

In
```
f(2, a=4, b=5)
```

Out
```
14
```

Coding Style and PEP8

To learn more about the Python programming philosophy type `import this` at the prompt

Among other things, Python strongly favors consistency in programming style

We've all heard the saying about consistency and little minds

In programming, as in mathematics, the opposite is true

- A mathematical paper where the symbols $\cup$ and $\cap$ were reversed would be very hard to read, even if the author told you so on the first page

In Python, the standard style is set out in PEP8

(Occasionally we'll deviate from PEP8 in these lectures to better match mathematical notation)

Exercises

Solve the following exercises

(For some, the built in function `sum()` comes in handy)

Exercise 1

Part 1: Given two numeric lists or tuples `x_vals` and `y_vals` of equal length, compute their inner product using `zip()`

Part 2: In one line, count the number of even numbers in 0,…,99

- Hint: `x % 2` returns 0 if `x` is even, 1 otherwise

Part 3: Given `pairs = ((2, 5), (4, 2), (9, 8), (12, 10))`, count the number of pairs `(a, b)` such that both `a` and `b` are even

Exercise 2

Consider the polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots a_n x^n = \sum_{i=0}^{n} a_i x^i \tag{1}$$

Write a function `p` such that `p(x, coeff)` that computes the value in (1) given a point `x` and a list of coefficients `coeff`

Try to use `enumerate()` in your loop

Exercise 3

Write a function that takes a string as an argument and returns the number of capital letters in the string

Hint: `'foo'.upper()` returns `'FOO'`

Exercise 4

Write a function that takes two sequences `seq_a` and `seq_b` as arguments and returns `True` if every element in `seq_a` is also an element of `seq_b`, else `False`

- By "sequence" we mean a list, a tuple or a string
- Do the exercise without using sets and set methods

Exercise 5

When we cover the numerical libraries, we will see they include many alternatives for interpolation and function approximation

Nevertheless, let's write our own function approximation routine as an exercise

In particular, without using any imports, write a function `linapprox` that takes as arguments

- A function `f` mapping some interval $[a, b]$ into $\mathbb{R}$
- two scalars `a` and `b` providing the limits of this interval
- An integer `n` determining the number of grid points
- A number `x` satisfying `a <= x <= b`

and returns the piecewise linear interpolation of `f` at `x`, based on `n` evenly spaced grid points `a = point[0] < point[1] < ... < point[n-1] = b`

Aim for clarity, not efficiency

Solutions

Exercise 1

Part 1 solution:

Here's one possible solution

In
```
x_vals = [1, 2, 3]
y_vals = [1, 1, 1]
sum([x * y for x, y in zip(x_vals, y_vals)])
```

Out
```
6
```

This also works

In
```
sum(x * y for x, y in zip(x_vals, y_vals))
```

Out
```
6
```

Part 2 solution:

One solution is

In
```
sum([x % 2 == 0 for x in range(100)])
```

Out
```
50
```

This also works:

In
```
sum(x % 2 == 0 for x in range(100))
```

Out
```
50
```

Some less natural alternatives that nonetheless help to illustrate the flexibility of list comprehensions are

In
```
len([x for x in range(100) if x % 2 == 0])
```

Out
```
50
```

and

In
```
sum([1 for x in range(100) if x % 2 == 0])
```

Out
```
50
```

Part 3 solution

Here's one possibility

In
```
pairs = ((2, 5), (4, 2), (9, 8), (12, 10))
sum([x % 2 == 0 and y % 2 == 0 for x, y in pairs])
```

Out
```
2
```

Exercise 2

In
```
def p(x, coeff):
    return sum(a * x**i for i, a in enumerate(coeff))
```

In
```
p(1, (2, 4))
```

Out
```
6
```

Exercise 3

Here's one solution:

In
```
def f(string):
    count = 0
    for letter in string:
        if letter == letter.upper() and letter.isalpha():
            count += 1
    return count
f('The Rain in Spain')
```

Out
```
3
```

Exercise 4

Here's a solution:

In
```
def f(seq_a, seq_b):
    is_subset = True
    for a in seq_a:
        if a not in seq_b:
            is_subset = False
    return is_subset

# == test == #

print(f([1, 2], [1, 2, 3]))
print(f([1, 2, 3], [1, 2]))
```

Out
```
True
False
```

Of course if we use the `sets` data type then the solution is easier

In
```
def f(seq_a, seq_b):
    return set(seq_a).issubset(set(seq_b))
```

Exercise 5

In
```
def linapprox(f, a, b, n, x):
    """
    Evaluates the piecewise linear interpolant of f at x on the interval
    [a, b], with n evenly spaced grid points.

    Parameters
    ============
        f : function
            The function to approximate

        x, a, b : scalars (floats or integers)
            Evaluation point and endpoints, with a <= x <= b

        n : integer
            Number of grid points

    Returns
    =========
        A float. The interpolant evaluated at x

    """
    length_of_interval = b - a
    num_subintervals = n - 1
    step = length_of_interval / num_subintervals

    # === find first grid point larger than x === #
    point = a
    while point <= x:
        point += step

    # === x must lie between the gridpoints (point - step) and point === #
    u, v = point - step, point

    return f(u) + (x - u) * (f(v) - f(u)) / (v - u)
```