Python Essentials

This notebook offers an introduction to the fundamental concepts and techniques employed in Python economics programming for students at the intermediate macroeconomic level. It also provides a valuable resource for students at all levels, offering insights into new knowledge.

Overview

Now let's cover some core features of Python in a more systematic way.

This approach is less exciting but helps clear up some details.

Data Types

Computer programs typically keep track of a range of data types.

For example, `1.5` is a floating point number, while `1` is an integer.

Programs need to distinguish between these two types for various reasons.

One is that they are stored in memory differently.

Another is that arithmetic operations are different

- For example, floating point arithmetic is implemented on most machines by a specialized Floating Point Unit (FPU).

In general, floats are more informative but arithmetic operations on integers are faster and more accurate.

Python provides numerous other built-in Python data types, some of which we've already met

- strings, lists, etc.

Let's learn a bit more about them.

Primitive Data Types

Boolean Values

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

In [ ]:
```
x = True
x
```

Out[ ]: True

We can check the type of any object in memory using the `type()` function.

In [ ]:
```
type(x)
```

Out[ ]: bool

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

Numeric Types

Numeric types are also important primitive data types.

We have seen `integer` and `float` types before.

Complex numbers are another primitive data type in Python

In [ ]:
```
x = complex(1, 2)
y = complex(2, 1)
print(x * y)
```

```
type(x)
```

5j

Out[ ]:
```
complex
```

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
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/Users/cheney_gao/Desktop/Intermediate Macroeconomics/python_essentials.ipynb Cell 25 line 2
      <a href='vscode-notebook-cell:/Users/cheney_gao/Desktop/Intermediate%20Macroeconomics/python_essentials.ipynb#X34sZmlsZQ%3D%3D?line=0'>1</a> x = (1, 2)
----> <a href='vscode-notebook-cell:/Users/cheney_gao/Desktop/Intermediate%20Macroeconomics/python_essentials.ipynb#X34sZmlsZQ%3D%3D?line=1'>2</a> 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

Tuple unpacking is convenient and we'll use it often.

Slice Notation

To access multiple elements of a sequence (a list, a tuple or a string), you can use Python's slice notation.

For example,

In [ ]:
```
a = ["a", "b", "c", "d", "e"]
a[1:]
```

Out[ ]: ['b', 'c', 'd', 'e']

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

Out[ ]: ['b', 'c']

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[ ]: ['d', 'e']

You can also use the format `[start:end:step]` to specify the step

In [ ]:
```
a[::2]
```

Out[ ]:
```
['a', 'c', 'e']
```

Using a negative step, you can return the sequence in a reversed order

In [ ]:
```
a[-2::-1] # Walk backwards from the second last element to the first elem
```

Out[ ]:
```
['d', 'c', 'b', 'a']
```

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[ ]: '/Users/cheney_gao/Desktop/Intermediate Macroeconomics'

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)
```

```
Testing
Testing again
```

In fact, the recommended approach in modern Python is to use a `with` statement to ensure the files are properly acquired and released.

Containing the operations within the same block also improves the clarity of your code.

> **Note**

> This kind of block is formally referred to as a *context*.

Let's try to convert the two examples above into a `with` statement.

We change the writing example first

In [ ]:
```
with open('newfile.txt', 'w') as f:
    f.write('Testing\n')
    f.write('Testing again')
```

Note that we do not need to call the `close()` method since the `with` block will ensure the stream is closed at the end of the block.

With slight modifications, we can also read files using `with`

In [ ]:
```
with open('newfile.txt', 'r') as fo:
    out = fo.read()
    print(out)
```

```
Testing
Testing again
```

Now suppose that we want to read input from one file and write output to another. Here's how we could accomplish this task while correctly acquiring and returning resources to the operating system using `with` statements:

In [ ]:
```
with open("newfile.txt", "r") as f:
    file = f.readlines()
    with open("output.txt", "w") as fo:
        for i, line in enumerate(file):
            fo.write(f'Line {i}: {line} \n')
```

The output file will be

In [ ]:
```
with open('output.txt', 'r') as fo:
    print(fo.read())
```

```
Line 0: Testing

Line 1: Testing again
```

We can simplify the example above by grouping the two `with` statements into one line

In [ ]:
```
with open("newfile.txt", "r") as f, open("output2.txt", "w") as fo:
        for i, line in enumerate(f):
            fo.write(f'Line {i}: {line} \n')
```

The output file will be the same

In [ ]:
```
with open('output2.txt', 'r') as fo:
    print(fo.read())
```

```
Line 0: Testing

Line 1: Testing again
```

Suppose we want to continue to write into the existing file instead of overwriting it.

we can switch the mode to `a` which stands for append mode

In [ ]:
```
with open('output2.txt', 'a') as fo:
    fo.write('\nThis is the end of the file')
```

In [ ]:
```
with open('output2.txt', 'r') as fo:
    print(fo.read())
```

```
Line 0: Testing

Line 1: Testing again

This is the end of the file
```

> **Note**
>
> Note that we only covered `r`, `w`, and `a` mode here, which are the most commonly used modes.

Python provides a variety of modes that you could experiment with.

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 be 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 [ ]:
```
%%writefile 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
```

```
Writing us_cities.txt
```

Here `%%writefile` is an IPython cell magic.

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

The program below 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()
```

```
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 `data_file` 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)
```

```
1
4
9
```

is preferred to

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

```
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}')
```

```
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}'")
```

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

List Comprehensions

We can also simplify the code for generating the list of random draws considerably by using something called a *list comprehension*.

List comprehensions are an elegant Python tool for creating lists.

Consider the following example, where the list comprehension is on the right-hand side of the second line

In [ ]:
```
animals = ['dog', 'cat', 'bird']
plurals = [animal + 's' for animal in animals]
plurals
```

Out[ ]:
```
['dogs', 'cats', 'birds']
```

Here's another example

In [ ]:
```
range(8)
```

Out[ ]:
```
range(0, 8)
```

In [ ]:
```
doubles = [2 * x for x in range(8)]
doubles
```

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

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`

We can also use `all()` and `any()` to test a sequence of expressions

In [ ]:
```
all([1 <= 2 <= 3, 5 <= 6 <= 7])
```

Out[ ]: True

In [ ]:
```
all([1 <= 2 <= 3, "a" in "letter"])
```

Out[ ]: False

In [ ]:
```
any([1 <= 2 <= 3, "a" in "letter"])
```

Out[ ]: True

> **Note**
>
> - `all()` returns `True` when *all* boolean values/expressions in the sequence are `True`

- `any()` returns `True` when *any* boolean values/expressions in the sequence are `True`

Coding Style and Documentation

A consistent coding style and the use of documentation can make the code easier to understand and maintain.

Python Style Guidelines: PEP8

You can find Python programming philosophy by typing `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 ∪ and ∩ 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)

Docstrings

Python has a system for adding comments to modules, classes, functions, 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?
```

```
Signature: f(x)
Docstring: This function squares its argument
File:      /var/folders/v4/8yvlmdh17719kc18c21wy27r0000gn/T/ipykernel_8008
6/238177437.py
Type:      function
```

```
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??
```

```
Signature: f(x)
Source:
def f(x):
    """
    This function squares its argument
    """
    return x**2
File:      /var/folders/v4/8yvlmdh17719kc18c21wy27r0000gn/T/ipykernel_8008
6/238177437.py
Type:      function
```

```
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.

You can find conventions for docstrings in PEP257.