Topics in Computational Economics

Lecture 4

John Stachurski

NYU 2016

Today's Lecture

- Data types
- Iteration
- Functions
- Namespaces
- OOP

Data Types

We have already met several native Python data types

```
In [1]: s = 'foo'
In [2]: type(s)
Out[2]: str

In [3]: y = 100
In [4]: type(y)
Out[4]: int

In [5]: x = 0.1
In [6]: type(x)
Out[6]: float
```

Type is important in Python

```
>>> 10 + '10'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 's
```

We say that Python is **strongly typed**

Container Types

Some data types contain other objects:

- lists
- tuples
- dictionaries
- sets

These are called **container types**

Tuples are similar to lists

```
In [1]: x = ['a', 'b']  # Square brackets for lists

In [2]: type(x)
Out[2]: list

In [3]: x = ('a', 'b')  # Round brackets for tuples

In [4]: type(x)
Out[4]: tuple

In [5]: x = 'a', 'b'    # No brackets is identical

In [6]: type(x)
Out[6]: tuple
```

In fact tuples are **immutable** lists

Immutable means internal state cannot be altered

```
>>> x = (1, 2)  # Tuples are immutable
>>> x[0] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Tuple and list unpacking:

```
In [1]: x = [10, 20]

In [2]: a, b = x

In [3]: a
Out[3]: 10

In [4]: b
Out[4]: 20
```

- Same works for `x = (10, 20)` or `x = 10, 20`

Example. How can we return multiple values from a function?

Answer: We can't — instead we use a tuple

Ex. Write a function that

1. takes as input a string of the form `john_smith`
2. returns the first name `John` and the surname `Smith`

Hint: Look at the set of string methods

Solution:

```
def split_name(firstname_lastname):
    """
    Separate john_smith into John, Smith
    """
    firstname, lastname = firstname_lastname.split('_')

    firstname = firstname.capitalize()
    lastname = lastname.capitalize()

    return firstname, lastname  # Returns tuple
```

What it returns:

```
In [2]: split_name('john_stachurski')
Out[2]: ('John', 'Stachurski')
```

Standard usage:

```
In [3]: first, last = split_name('john_stachurski')

In [4]: first
Out[4]: 'John'

In [5]: last
Out[5]: 'Stachurski'
```

Another example of a function that returns a tuple:

```
>>> from scipy.stats import norm
>>> from scipy.integrate import quad
>>> phi = norm()
>>> value, error = quad(phi.pdf, -2, 2)  # Returns tuple
>>> value
0.9544997361036417
```

Dictionaries are similar to lists, items named instead of numbered

```
In [1]: d = {'name': 'Frodo', 'age': 33}

In [2]: type(d)
Out[2]: dict

In [3]: d['name']
Out[3]: 'Frodo'

In [4]: d.keys()
Out[4]: dict_keys(['age', 'name'])

In [5]: d.values()
Out[5]: dict_values([33, 'Frodo'])
```

Of course there are also many third party types

```
In [1]: import numpy as np

In [2]: a = np.random.randn(400)

In [3]: type(a)
Out[3]: numpy.ndarray

In [4]: a.min()
Out[4]: -2.7496909420190905
```

We'll learn to make our own soon

Input and Output

Let's start with writing

```
In [1]: f = open('newfile.txt', 'w')

In [2]: f.write('Testing\n')

In [3]: f.write('Testing again\n')

In [4]: f.close()
```

Files are written to present working directory (`pwd` in IPython)

To read the contents of `newfile.txt`:

```
In [5]: f = open('newfile.txt', 'r')

In [6]: out = f.read()

In [7]: print(out)
Testing
Testing again

In [8]: f.close()
```

File must be in pwd (otherwise specify full path)

Iterating

```
animals = ['dog', 'cat', 'bird']
for animal in animals:
    print("One " + animal + ", two " + animal + "s")
```

Output:

```
One dog, two dogs
One cat, two cats
One bird, two birds
```

Example: The file `us_cities.txt` looks as follows

```
new york: 8244910
los angeles: 3819702
chicago: 2707120
houston: 2145146
philadelphia: 1536471
phoenix: 1469471
san antonio: 1359758
san diego: 1326179
dallas: 1223229
```

We want to clean it up like so:

```
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
```

Solution

```
data_file = open('us_cities.txt', 'r')
for line in data_file:
    city, population = line.split(':')
    city = city.title()
    population = '{0:,}'.format(int(population))
    print(city.ljust(15) + population)
data_file.close()
```

Under the hood: file objects are **iterators**

- An object that implements a `__next__` method

```
In [1]: f = open('us_cities.txt')

In [2]: f.__next__()
Out[2]: 'new york: 8244910\n'

In [3]: f.__next__()
Out[3]: 'los angeles: 3819702\n'

In [4]: f.__next__()
Out[4]: 'chicago: 2707120 \n'
```

```
In [1]: x = ['dog', 'cat', 'bird']  # Iterable

In [2]: x = iter(x)  # iter() creates an iterator

In [3]: x.__next__()
Out[3]: 'dog'

In [4]: x.__next__()
Out[4]: 'cat'

In [5]: x.__next__()
Out[5]: 'bird'
```

List Comprehensions

```
>>> animals = ['dog', 'cat', 'bird']
>>> plurals = [animal + 's' for animal in animals]
>>> plurals
['dogs', 'cats', 'birds']
```

```
>>> evens = [i for i in range(10) if i % 2 == 0]
>>> evens
[0, 2, 4, 6, 8]
```

Ex. Get a copy of this file

- https://github.com/QuantEcon/QuantEcon.applications/blob/master/python_essentials/us_cities.txt

Write a Python program that

1. Reads in the data from the file `us_cities.txt`
2. Prints the sum of the populations of the cities

Output should be 23,831,986

Solution:

```
total = 0
f = open('us_cities.txt')
for line in f:
    city, pop = line.split(':')
    total += int(pop)
print("Total = {0:,}".format(total))
f.close()
```

Functions

Some are built-in:

```
>>> max(19, 20)
20
>>> type(max)
<class 'builtin_function_or_method'>
```

Others are imported:

```
from math import sqrt
```

We can also write our own functions

```
def f(x):
    return x + 42
```

One line functions using the `lambda` keyword:

```
f = lambda x: x + 42
```

A common use of `lambda`

To calculate $\int_0^2 x^3 dx$ we can use SciPy's `quad` function

Syntax is `quad(f, a, b)` where

- `f` is a function and
- `a` and `b` are numbers

```
>>> from scipy.integrate import quad
>>> quad(lambda x: x**3, 0, 2)
(4.0, 4.440892098500626e-14)
```

Python functions are flexible:

```
>>> f = lambda x: 2 * x
>>> f(2)
4
>>> g = f
>>> g(2)
4
>>> h = (f, g)
>>> h[0](2)
4
```

Functions can create and return **nested functions**

```
def f_creator(a, r):
    """
    Create a CES production function f with
    parameters a, r
    """
    r_inv = 1 / r
    def f(k, ell):
        return (a * k**r * (1 - a) * ell**r)**r_inv
    return f
```

Usage:

```
In [2]: f1 = f_creator(0.5, 0.5)

In [3]: f1(2, 3)
Out[3]: 0.37500000000000006

In [4]: f2 = f_creator(0.4, 0.6)

In [5]: f2(2, 3)
Out[5]: 0.5561218855328426
```

Names and Namespaces

Consider this assignment statement

```
x = 42
```

We are **binding** the **name** `x` to the object on the right hand side

Thus, **names are symbols bound to objects stored in memory**

Python is **dynamically typed** — names are not specific to type

```
s = 'foo'   # Bind s to a string
s = 42      # and now to an integer
```

Consider this (frightening?) code example

```
In [1]: x = ['foo', 'bar']

In [2]: y = x

In [3]: y[0] = 'fee'

In [4]: x
Out[4]: ['fee', 'bar']
```

Works because

- x and y are bound to the **same object**
- that object is **mutable**

Without mutability we get different behavior

```
In [5]: x = 1

In [6]: y = x

In [7]: y = 2  # Binds y to a new object

In [8]: x
Out[8]: 1
```

Namespaces

Problem: Suppose I assign a variable `path` like so

```
>>> path = '/home/john'
```

Now I do this

```
>>> from os import *
```

But now my variable `path` is **shadowed** by `path` from `os`

```
>>> path
<module 'posixpath' from ...>
```

How to avoid name conflicts?

Python addresses this problem using namespaces

A **namespace** is a **mapping** from names to Python objects

Python uses multiple namespaces to give names context

For example, modules have their own namespace

```
>>> import sys
>>> sys.path
['/home/john/bin', ...]

>>> import os
>>> os.path
<module 'posixpath' from ...>
```

```
>>> import math
>>> vars(math)  # Print namespace

{'atan': <built-in function atan>,
'cos': <built-in function cos>,
'floor': <built-in function floor>,
'pow': <built-in function pow>,...}
```

- (output was exited for readability)

Implementation: Namespaces are stored in dictionaries

```
>>> import math
>>> math.__dict__  # View the namespace directly

{'atan': <built-in function atan>,
'cos': <built-in function cos>,
'floor': <built-in function floor>,
'pow': <built-in function pow>,...}
```

Interactive sessions execute in a module called `__main__`

Let's look at its namespace

```
>>> vars()
{'__name__': '__main__',
'__spec__': None,
..., '__doc__': None}

>>> x = 'foobar!!'
>>> vars()
{'__name__': '__main__',
'__spec__': None,
..., '__doc__': None,
'x': 'foobar!!'}
```

When functions are invoked they get their own namespace

```
def f(x):
    a = 2
    print("local names:", locals())
```

Calling the function produces the following output

```
>>> f(3)
local names: {'a': 2, 'x': 3}
```

Name Resolution

Consider this code

```python
x = 4
def f():
    x = 5
    print(x)
```

Running this gives the following

```
In [33]: f()
5
In [34]: x
Out[34]: 4
```

How does it work?

The order in which the interpreter searches for names is

1. the local namespace (if exists)
2. the hierarchy of enclosing namespaces (if exist)
3. the global namespace
4. the builtin namespace

If the name is not in any of these namespaces, the interpreter raises a `NameError`

Called the LEGB rule

Object Oriented Programming

Traditional programming paradigm is called **procedural**

- A program has state (values of its variables)
- Functions are called to act on this state
- Data is passed around via function calls

In **OOP**, data and functions bundled together into **objects**

These bundled functions are called **methods**

Example: Lists = list data + list methods

```
>>> x = [1, 5, 4]
>>> x.append(7)
>>> x
[1, 5, 4, 7]
```

Compare with Julia:

```
julia> x = Any[]
0-element Array{Any,1}

julia> push!(x, "foo")
1-element Array{Any,1}:
 "foo"
```

```
from envelopes import Envelope

envelope = Envelope(
    from_addr=’from@example.com’,
    to_addr=’to@example.com’,
    subject=’Envelopes demo’,
    text_body="I’m a helicopter!")

envelope.add_attachment(’/Users/bilbo/helicopter.jpg’)
envelope.send(’smtp.googlemail.com’,
              login=’from@example.com’,
              password=’password’,
              tls=True)
```

Why OOP?

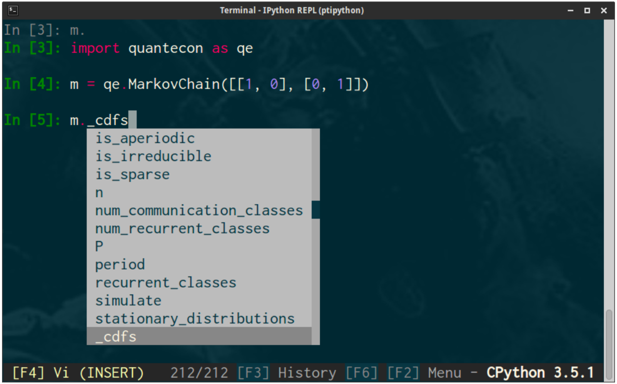
Economize on global names

```
>>> x = ['foo', 'bar']
>>> x.append('fee')
```

The alternative would be `append(x, 'fee')`

But then you need another function in the global namespace

Advantage number 2: introspection



Fits human experience — many entities combine data and actions

```
class Economist:

    data:
        alma mater
        publications

    methods:
        research
        teach
        eat
        sleep
        make jokes about economics
```

Python's Approach to OOP

Python is **partly** object oriented

- Everything is an object
- Native data types have methods
- Easy to build new objects bundling data and methods

But Python also uses ordinary functions

- `x.append(val)` but `max(x)`

Everything is an object

Internally, everything is an object

```
>>> x = 1
>>> dir(x)
['__abs__',
 '__add__',
 .
 .
 'denominator',
 'imag',
 'numerator',
 'real']
```

Each Python object has a type, id, value, zero or more methods

```
>>> x = 1

>>> type(x)
<type 'int'>

>>> id(x)
10325096

>>> x.__add__(1)
2
```

Building New Classes of Objects

We can define our own classes with the `class` keyword

```
class Foo:
    "This class does nothing"
    pass
```

Create instances by a function call on the class name

```
>>> from foo_def import Foo  # def in file foo_def.py
>>> foo1 = Foo()
>>> foo2 = Foo()
>>> type(foo1)
<class 'foo_def.Foo'>
```

A class is really just a fancy namespace

```
class Foo:
    "Another useless class"
    some_var = 1
```

```
>>> Foo.__dict__
{'some_var': 1,
'__module__': '__main__',
'__doc__': 'Another useless class'}
>>> foo = Foo()
>>> foo.some_var
1
```

For this class

```python
class Foo:
    "Another useless class"
    some_var = 1
```

all data is class data (not instance data)

```python
>>> foo_1 = Foo()
>>> foo_2 = Foo()
>>> foo_1.some_var is foos_2.some_var
True
```

Usually we want instances of the class to have their own data too

Imagine cars in a race game

We want cars to share some structure (defined in a class)

But we want them to have their own

- location
- speed
- color, etc.

We can give instances their own data using the `self` keyword

To illustrate, let's build a class to represent die

Here's the pseudocode

```
class Dice:

    instance data:
        face -- the side facing up

    methods:
        roll -- roll the dice and change face
```

Here's how the actual code in file `dice.py`

```
import random

possible_faces = (1, 2, 3, 4, 5, 6)

class Dice:

    def __init__(self):
        self.face = 1

    def roll(self):
        self.face = random.choice(possible_faces)
```

Let's try it

```
In [7]: run dice.py

In [8]: d1 = Dice()

In [9]: d1.face
Out[9]: 1

In [10]: d1.roll()

In [11]: d1.face
Out[11]: 2
```

Instance data lives in a dictionary called `__dict__`

```
In [7]: d1 = Dice()

In [9]: d1.__dict__
Out[9]: {'face': 1}

In [10]: d1.roll()

In [11]: d1.face
Out[11]: 2

In [12]: d1.__dict__
Out[12]: {'face': 2}
```

We can create as many die as we want

Each has

- their own namespace
- their own instance data, which their methods act on

Ex. Generate 100,000 die

- All exist in memory at the same time

Roll each one at least once

Count the fraction of die with an even face

One solution:

```python
from dice import Dice

n = 100000
die = [Dice() for i in range(n)]

mean = 0
for dice in die:
    dice.roll()
    if dice.face in (2, 4, 6):
        mean +=1

print(mean / n)
```

Behind the Scenes

Consider again the definition of the roll method

```
def roll(self):
    self.face = random.choice(possible_faces)
```

The actual reason `self` is in the method is that

```
In [19]: d.roll()
```

is in fact translated into the call

```
In [20]: Dice.roll(d)
```