Leiden University
Medical Center

Python Programming

Flow Control

Mihai Lefter

Outline

Working with scripts

Interpreters are great for prototyping, but not really suitable if you want to share or release code. To do so, we write our Python commands in scripts (and later, modules). A script is a simple text file containing Python instructions to execute.

Executing scripts

There are two common ways to execute a script:

- As an argument of the Python interpreter command.
- As a standalone executable (with the appropriate shebang line and file mode).

IPython gives you a third option:

- As an argument of the `%run` magic.

Writing your script

Let's start with a simple hello world example.

Open your text editor and write the following Python statement:

first_script.py

```
1 print("Hello world!")
```

Save the file as first_script.py and go to your shell.

Running the script

Let's try the first method, i.e., using your script as an argument:

terminal

```
$ python first_script.py
```

Is the output as you expect?

Running the script

For the second method, we need to do two more things:

- Open the script in your editor and add the following line to the very top:
  - *#!/usr/bin/env python*
- Save the file, go back to the shell, and allow the file to be executed.

terminal

```
$ chmod +x first_script.py
```

You can now execute the file directly:

terminal

```
$ ./first_script.py
```

Is the output the same as the previous method?

Running the script

Finally, try out the third method. Open an IPython interpreter session and do:

IPython

```
In [1]: %run first_script.py
```

if statements

if_example.py

```
1  if 26 <= 17:
2      print('Fact: 26 is less than or equal to 17')
3  elif (26 + 8 > 14) == True:
4      print("Did we need the ' == True' part here?")
5  else:
6      print('Nothing seems true')
```

terminal

```
$ python if_example.py
Did we need the ' == True' part here?
```

while statements

while_example.py

```
1 i = 0
2 while i < 5:
3     print(i)
4     i += 1
```

terminal

```
$ python while_example.py
0
1
2
3
4
```

Iterating over a sequence

for_example.py

```
1  colors = ['red', 'white', 'blue', 'orange']
2  cities = ['leiden', 'utrecht', 'warmond', 'san francisco']
3
4  # The for statement can iterate over sequence items.
5  for color in colors:
6      print(color)
7
8  for character in 'blue':
9      print(character)
```

Iterating over a sequence

terminal

```
$ python for_example.py
red
white
blue
orange
b
l
u
e
```

Python anti-patterns

These are common for programmers coming from other languages.

unpythonic.py

```
1  i = 0
2  while i < len(colors):
3      print(colors[i])
4      i += 1
5
6  for i in range(len(colors)):
7      print(colors[i])
```

We call them unpythonic.

Additionals

iteration.py

```
# Iteration with values and indices:
for i, color in enumerate(colors):
    print(i, '->', color)

# Taking two sequences together:
for city, color in zip(cities, colors):
    print(city, '->', color)

# Iterating over a dictionary yields keys:
for key in {'a': 33, 'b': 17, 'c': 18}:
    print(key)

# Iterating over a file yields lines:
for line in open('data/short_file.txt'):
    print(line)
```

Indentation

Python uses indentation to delimit blocks

- Instead of `begin ... end` or { ... } in other languages.
- Always increase indentation by 4 spaces, never use tabs.
  - In any case, be consistent.

indentation_example.py

```
1 if False:
2     if False:
3         print('Why am I here?')
4     else:
5         while True:
6             print('When will it stop?')
7     print("And we're back to the first indentation level")
```

Comments

Comments are prepended by # and completely ignored.

comments_example.py

```
1 # Create the list.
2 l = []
3
4 # Add 42 to this list.
5 l.append(42)
```

The pass statement

If you ever need a statement syntactically but don't want to do anything, use `pass`.

comments_example.py

```
while False:
    # This is never executed anyway.
    pass
```

Defining a function

A function is a named sequence of statements that performs some piece of work. Later on that function can be called by using its name.

A function definition includes its name, arguments and body.

functions.py

```
1  def add_two(number):
2      return number + 2
3
4  for i in range(5):
5      print(add_two(i))
```

Keyword arguments

Besides regular arguments, functions can have keyword arguments.

functions_keywords.py

```
1  def add_some_other_number(number, other_number=12):
2      return number + other_number
3
4  add_some_other_number(2, 6)  # 8
5
6  add_some_other_number(3, other_number=4)  # 7
7
8  add_some_other_number(5)  # 17
```

Functions are values

We can pass functions around just like other values, and call them.

function_values.py

```
1  def add_two(number):
2      return number + 2
3
4  def add_some_other_number(number, other_number=12):
5      return number + other_number
6
7  functions = [add_two, add_some_other_number]
8  for function in functions:
9      print(function(7))
10
11 # Simple anonymous functions can be created with lambda.
12 functions.append(lambda x: x * 7)
13 for function in functions:
14     print(function(4))
```

Docstrings

Like many other definitions, functions can have docstrings.

- Docstrings are regular string values which you start the definition body with.
- You can access an object's docstring using help.

docstring_example.py

```
1  def factorial(n):
2      """Compute factorial of n in the obious way."""
3      if n == 0:
4          return 1
5      else:
6          return factorial(n - 1) * n
```

Higher-order functions

A function that takes a function as argument is a higher-order function.

IPython

```
In [2]: help(map)
        Help on class map in module builtins:
        class map(object)
         |  map(func, *iterables) --> map object
         |
         |  Make an iterator that computes the function using arguments from
         |  each of the iterables.  Stops when the shortest iterable is
        ↪   exhausted.

In [3]: list(map(add_two, [1, 2, 3, 4]))
Out[3]: python [3, 4, 5, 6]
```

List comprehensions

Similar to mathematical set notation (e.g., $x|x \in R \wedge x > 0$), we can create lists.

IPython

```
In [4]:  [(x, x * x) for x in range(10) if x % 2]

Out[4]:  python [(1, 1), (3, 9), (5, 25), (7, 49), (9, 81)]
```

We can do the same thing using `map` and `filter`, but list comprehensions are often more readable.

IPython

```
In [5]:  list(map(lambda x: (x, x * x), filter(lambda x: x %2, range(10))))

Out[5]:  python [(1, 1), (3, 9), (5, 25), (7, 49), (9, 81)]
```

Set and dictionary comprehensions

Similar notation can be used for (non-empty) sets.

IPython

```
In [6]:  {c for c in 'LUMC-standard' if 'a' <= c <= 'z'}

Out[6]:  python 'a', 'd', 'n', 'r', 's', 't'
```

We can do the same thing using `map` and `filter`, but list comprehensions are often more readable.

IPython

```
In [7]:  colors = ['red', 'white', 'blue', 'orange']
In [8]:  {c: len(c) for c in colors}

Out[8]:  python 'blue':  4, 'orange':  6, 'red':  3, 'white':  5
```

Hands on!

1. Write a Python function that returns the maximum of two numbers.
2. Write a Python function that returns the maximum of three numbers. Try to reuse the first maximum of two numbers function.
3. Write a Python function that accepts a string as parameter. Next, it calculates and prints the number of upper case letters and lower case letters. Make use of the `isupper` and `islower` built in methods.

Leiden University Medical Center

Acknowledgements

Martijn Vermaat
Jeroen Laros
Jonathan Vis