Basic Python Programming

Yilin Zheng

Dept. of Computer Science and Engineering

July 2, 2017

Outline

Define a function

- Use `def` **a function name (parameters):** to define a function
- Form:

```
def Function_Name(parameters):
    #Code blocks
    return
```

Parameters

- Parameters are **specified in parentheses** and separated by **comma ,**
- Pass value **in corresponding order** when calling functions

Default parameter value

- Default parameter value **cannot be modified**, and **can only be set in the last**
- Use `=` with **the default value**

Key parameter

- Use **parameter name** with =
- Depends on **the names of parameters but not their positions**

Local variable

- Local variables are locally valid, have **the scope within functions**
- Identifiers can **be the same name with external variables**

global

- Add `global` before the identifiers so as to make variables have global scope

return

- Use `return` to **indicate the end of function** or **return results**
- If no `return`, functions will return `None`
- Use `pass` to represent an **empty statement block**

DocStrings

- Fixed position, **the first expression within the function** and specified in ''' '''
- **First line** begins with **capital letter** and ends with **period** .
- **Second line** leaves **empty**
- **Third line and later lines** are used to **describe the detail**
- Use `.__doc__` (**double underscore**) or built-in function `help` to refer

Modules

- Some functions and variables will be reused for many times, so put them together in a **.py** file, called **modules**
- Basically, modules include **functions** and **variables**
- Some common modules: sys, math, json, requests, XML...
- We usually import modules we need in order to reuse the codes, it saves time and codes

sys

- Let's take "sys" as an example
- This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter, more on ▸ sys for 3.6.2rc1

```
#!/usr/bin/python
# Filename: print_sys.py
import sys #import this module
print('The command line arguments are:')
for i in sys.argv: #sys.argv contains command
                   line arguments
  print(i)
print('\nThe PYTHONPATH is', sys.path, '\n')
```

- Pay attention to the first expression, which is the way to use a module, but how it works?

How it works

- In the example last page, we use `import` **module name** to indicate we want to use that module
- When executing the codes, interpreter will find "sys.py" (in the directory listed by sys.path) and execute the codes from its main block
- Use **.variable name** to indicate the variables defined in the module

Byte-complied .pyc

- Import modules sometimes is arduous, so use byte-complied .pyc to accrelerate this process
- This kind of files ends with **.pyc** rather than .py
- It is related to the intermediate form that Python transforms the program into

from...import

- We have known how to use `import`, but what if we **only want to use some variables**?
- Use `from Module name import sth.` to import parts of variables or classes or functions within the module
- In previous case, if we only want to use sys.argv, then we can code `from sys import argv`, then we **don't need to type its module name** sys. every time referring to it
- Also, we can use `from Modules name import *` to import all that defined within the module
- By this way, we can make codes **more comprehensible**, and **avoid conflicts on names**

Example

- See what the different is

```
#!/usr/bin/python # Filename: print_pi.py
from math import pi
print(pi) # print the value of PI

#!/usr/bin/python # Filename: print_pi.py
import math
print(math.pi)
```

- These two blocks return the same result

__name__

- Every module has its own name, we can refer to the name by using `__name__`
- `__name__` stores the name of the current module

```
#!/usr/bin/python # Filename: print_name.py
import math
print(math.__name__) # it will print 'math'
```

Create own modules

- We can create our own modules by **using Python script**
- Use `import` or `from...import`
- Example: create a file named 'sing_a_song' and add the following codes

```
#!/usr/bin/python # Filename: sing_a_song.py
def sing_a_song():
    print("""I\'ll never let you see
The way my broken heart is hurting\' me
I\'ve got my pride and I know how to hide
All my sorrow and pain
I\'ll do my crying\' in the rain""")
```

Create own modules

- Create aonther file named 'test_module', add the following codes, save both files in the same directory

```
#!/usr/bin/python
# Filename: test_module.py
import sing_a_song
sing_a_song.sing_a_song()
```

- Execute 'test_module' in terminal and see the outcome
- You will also find a folder named '__pycache__' was generated in the same directory, with a .pyc file in it

Function: dir

- `dir` is a built-in function
- **Without arguments**, it can return **a list of names in current local scope**
- **With an arguments**, it returns **a list of valid attributes for that object**

```
>>> a = 10
>>> b = 20
>>> c = 'Good night'
>>> dir() # no arguments
['__annotations__', '__builtins__', '__doc__',
'__loader__', '__name__', '__package__',
'__spec__', 'a', 'b', 'c']
>>> dir(a) # an arguments
['__abs__', '__add__', '__and__', '__bool__',
'__ceil__', '__class__', '__delattr__', '
                                __dir__', ...
```