pue
IMPULSANDO EL CONOCIMIENTO
TIC CUALIFICADO

EDUCACIÓN
CALIDAD Y CERTIFICACIÓN
EN SUS PROGRAMAS EDUCATIVOS

Programación de Redes – Becas Digitaliza - 2019

PUE – ITC – Formación de Instructores

Sesión 3 – Introducción a Python (II)

Iván Lago - Técnico Cisco Networking Academy ASC/ITC
PUE - ITC/ASC/CA

Index

- Functions
- Read Files
- Modules and packages
- Errors and exceptions

Functions

- Print(), input(), int(), float()… -> Methods (functions)
- There are a lot of functions, and we can create them.
- Are they really needed?

Functions: why do we need them? (I)

- Sometimes we need a piece of code many times
  - Completely literally or with minor modifications.

Clone the code many times?

Functions: why do we need them? (II)

- Large and heavy programs usually need focus from many people

- Should everyone work in the same document or can they work separately?

Functions: the solution

Functions

- If one error is detected in a function there will be just one change (not propagated)

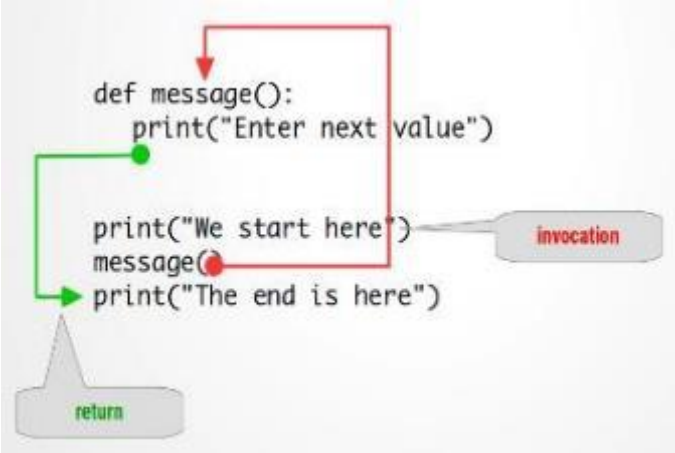
- The code can be decomposed and each programmer can work with his piece of code separately.

Functions: where they come from?

- **Python itself:** integral part of Python (print(), int()…)
- **Python preinstalled modules:** lot of useful functions used less than the often ones, are available in Python, but we have to import modules (we'll see it later)
- **Our own code:** we can write our own functions inside the code and use them as many as we want.

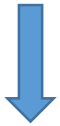
How to write a function?

- It always starts with the keyword ***def*** (for “define”)
- next after ***def*** goes the name of the function (the rules for naming functions are exactly the same as for naming variables)
- after the function name, there’s a place for a pair of parentheses (they contain nothing here, but that will change soon)
- the line has to be ended with a colon “***:***”
- the line directly after ***def*** begins the function body – a couple (at least one) of necessarily nested instructions, which will be executed every time the function is invoked; note: the function ends where the nesting ends, so you have to be careful



Functions: common errors

- **You mustn't invoke a function which is not known at the moment of invocation.**



- Remember – Python reads your code from top to bottom. It's not going to look ahead in order to find a function you forgot to put in the right place ("right" means "before invocation")

```
print("We start here")
message()
print("The end is here")

def message():
    print("Enter next value")
```

- **You mustn't have a function and a variable of the same name.**

```
def message():
    print("Enter next value")

message = 1
```

Parametrized functions (I)

- Functions can accept data provided by the invoker (data is provided as a parameter).

- A **parameter** is actually a variable, but there are two important factors that make parameters different and special:
  - they exist **only inside functions** in which they have been defined, and the only place where the parameter can be defined is a space between a pair of parentheses in the def statement;
  - assigning a value to the parameter is **done at the time of the function's invocation**, by specifying the corresponding argument.

- Don't forget:
  - **parameters** live inside functions (this is their natural environment)
  - **arguments** exist outside functions, and are carriers of values passed to corresponding parameters.

Parametrized functions (II)

- Function price with 2 parameters:
  - what (article)
  - no (price)
- Conditional
- Print with concatenates and casted values
- Invocation of the function

```
def price(what, no):
    if what == "usb":
        print("The price of", str(no),what, "is", str(5*no), "€")


price("usb", 10)
```

```
The price of 10 usb is 50 €
```

Even better 

```
def price(what,no):
    dict={"usb":10,"pc":100}
    print("The price of", str(no),what,"is:",str(no*dict[what]))

dict={"usb":10,"pc":100}
price("usb",5)
```

```
The price of 5 usb is: 50
```

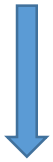
Parametrized functions: typical error

- You mustn’t use a non-existent parameter name

```
def introduction(firstname,lastname):
  print("Hello, my name is ",firstname,lastname,".")

introduction(name="Skywalker",firstname="Luke")
```

Parametrized function: order of parameters

```
def people(firstname="Luke", lastname="Skywalker"):
    print("The guy you are talking about is", firstname, lastname + ".")


people()
people("Luke")
people(firstname="Luke")
people("Harry")
people("Skywalker")
people(lastname="Potter")
people("Harry", "Potter")
people(lastname="Potter", firstname="Harry")
```

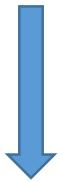


```
The guy you are talking about is Luke Skywalker.
The guy you are talking about is Luke Skywalker.
The guy you are talking about is Luke Skywalker.
The guy you are talking about is Harry Skywalker.
The guy you are talking about is Skywalker Skywalker.
The guy you are talking about is Luke Potter.
The guy you are talking about is Harry Potter.
The guy you are talking about is Harry Potter.
```

Returning result from a function

- If we want to get a value to use in the code from a function, we have to use the keyword ***return***.
- ***return*** terminates the function and return to the point of invocation.

```
def returning_function(value1, value2):
    return value1*value2


calculus = returning_function(5, 6)
print("The calculus is:", calculus)
```



```
The calculus is: 30
```

Functions: advanced

We can combine functions with loops, lists, dictionaries…

```
def function(list):
    print(list)
    del list[0]


L = [2, 3, 4]
print("The values stored in the list before the function are:", L)
function(L)
print("The values stored in the list after the function are:", L)
```

```
The values stored in the list before the function are: [2, 3, 4]
[2, 3, 4]
The values stored in the list after the function are: [3, 4]
```

Functions: scope (I)

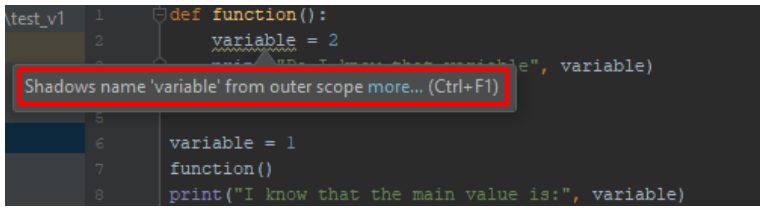
```
def function():
    print("Do I know that variable", variable)


variable = 1
function()
print("I know that the main value is:", variable)
```

```
Do I know that variable 1
I know that the main value is: 1
```

why?

```
def function():
    variable = 2
    print("Do I know that variable", variable)


variable = 1
function()
print("I know that the main value is:", variable)
```

```
Do I know that variable 2
I know that the main value is: 1
```



Functions: scope (II)

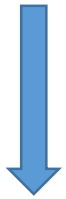
```
def function():
    global variable
    variable = 2
    print("The value in the function is:", variable)


variable = 1
print("Before the function invocation, the value is:", variable)
function()
print("After the invocation, the value is:", variable)
variable = 3
print("If we change the variable again, the value is:", variable)
```

```
Before the function invocation, the value is: 1
The value in the function is: 2
After the invocation, the value is: 2
If we change the variable again, the value is: 3
```

Modules: why do we need them?

- A way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. They are like extra equipment.

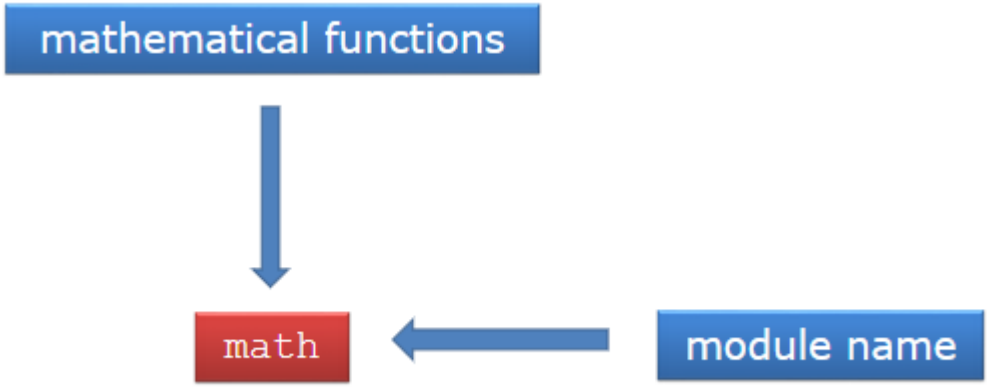
- Since the code is always growing, is so hard to handle all in one file, so we dice divide pieces of software into separates but cooperating files.



How?

Modules

- A module is identified by its name.
- Theses modules, along with the built-in functions, form the Python standard library (a special sort of library where modules play the roles of books).
- Each module have entities (like a book has chapters), such as functions, variables, constants, classes, and objects.



Modules: *import* (I)

- To make a module usable, we have to import it (like taking a book off the shelf) with the instruction: ***import <module>*** (i.e.: import math).

- In order to use the entities from a module, we have to put:
  - the name of the module
  - a dot
  - the name of the entity

- Examples: math.pi, math.sin…

```
import math
print(math.sin(math.pi/2))
```

Modules: *import* (II)

- Another form to import is:
  - ***from <module> import <entity>*** (i.e.: from math import pi)

- The instruction consists of the following elements:
  - the ***from*** keyword
  - the name of the module
  - the ***import*** keyword
  - the name of the entity/entities

```
from math import sin,pi

print(sin(pi/2))
```

Modules: *import* (III)

- Finally, we also can import all the entities of a module:
  - ***from <module> import ****
- It is **convenient** because it relieves us of the duty of enumerating all the names we need.
- It is unsafe because if we do not know all the name in a module, we may not be able to avoid name conflicts.
- We also can import modules and entities with an alias:
  - ***import <module> as <alias>***
  - ***from <module> import <entity> as <alias>***

Modules: *dir*

Reveal all the names provided through a particular module (it has to be previously imported as ***import <module>***)

```
import math

for name in dir(math):
  print(name, end='\t')
```

Modules: create module & import

module_test.py

```
#!/usr/bin/env python3

""" module_test.py is an example of modules"""

__counter = 0


def suml(list):
    global __counter
    __counter += 1
    sum = 0
    for el in list:
        sum += el
    return sum


def prodl(list):
    global __counter
    __counter += 1
    prod = 1
    for el in list:
        prod *= el
    return prod


if __name__ == "__main__":
    print("I prefer to be a module, but I can do some tests for you")
    l = [i+1 for i in range(5)]
    print(suml(l) == 15)
    print(prodl(l) == 120)
```

main.py

```
from module_test import suml, prodl


zeroes = [1 for i in range(5)]
ones = [2 for i in range(5)]
print(suml(zeroes))
print(prodl(ones))
```

Packages

Group of many modules (in the same way as we group functions in a module)

Errors

Codes can go wrong:

- Strings
- Negative values

```
import math
x = float(input("Enter x: "))
y = math.sqrt(x)
print("Square root of",x,"equals to",y)
```

Exceptions

- **try:** keyword to begin a block which may or may not be performing correctly
  - Nested code to be executed (the objective of the code)

- **except:** keyword to begin a block which is executed if the try block fails.
  - Nested code to be executed to prompt that there was an error

- **Useful exceptions:** ArithmeticError, AssertionError, BaseException, Exception, IndexError, KeyboardInterrupt, LookupError, MemoryError, OverflowError, ImportError, KeyError…

```
try:
    a = int(input("Tell me the value a: "))
    b = int(input("Tell me the value b: "))
    try:
        print(a / b)
    except:
        print("It cannot be done!")
except:
    print("Some value is not in the proper format")
print("The End")
```

Read an External File and Print the Contents

- One method to read a file is creating the script shown below.
- Remember that the file (in this case, 'devices.txt') should be in the same directory as your script.

```
file=open("devices.txt","r")
for item in file:
    print(item)
file.close()
```

```
import errno


try:
    stream = open("c:/Users/ivan/Documents/python_test.txt", "rt")
    for el in stream:
        print(el)
    stream.close()
except Exception as exc:
    if exc.errno == errno.ENOENT:
        print("The file does not exist")
    elif exc.errno == errno.EMFILE:
        print("You've opened too many files")
    else:
        print("The error number is", exc.errno)
```

Remove Blank Lines from the Output

Use strip attribute to remove the blank lines.

```
file=open("devices.txt","r")
for item in file:
    item=item.strip()
    print(item)
file.close()
```

```
import errno


try:
    stream = open("c:/Users/ivan/Documents/python_test.txt", "rt")
    for el in stream:
        el = el.strip()
        print(el)
    stream.close()
except Exception as exc:
    if exc.errno == errno.ENOENT:
        print("The file does not exist")
    elif exc.errno == errno.EMFILE:
        print("You've opened too many files")
    else:
        print("The error number is", exc.errno)
```

Copy File Content Into a List Variable

- Create an empty list.
- Use the **append** attribute to copy file content to the new list.

```
devices=[]
file=open("devices.txt","r")
for item in file:
    item=item.strip()
    devices.append(item)
file.close()
print(devices)
```

Read and write: creating different streams

One way to read, write or append information in a file is creating a stream for each function (it can be seen in the picture)

```
import errno


try:
    stream = open("c:/Users/ivan/Documents/python_test.txt", "r")
    data = stream.read()
    print(data)
    stream.close()
    stream = open("c:/Users/ivan/Documents/python_test.txt", "a")
    stream.write("\n\nAppend characters")
    stream.close()
except Exception as exc:
    if exc.errno == errno.ENOENT:
        print("The file does not exist")
    elif exc.errno == errno.EMFILE:
        print("You've opened too many files")
    else:
        print("The error number is", exc.errno)
```

Read and write: creating one stream

One alternative is just create one stream for read the file and, then, write information.

```
import errno


try:
    stream = open("c:/Users/ivan/Documents/python_test.txt", "r+")
    data = stream.read()
    print(data)
    stream.write("\n\nIt is okay")
    stream.close()
except Exception as exc:
    if exc.errno == errno.ENOENT:
        print("The file does not exist")
    elif exc.errno == errno.EMFILE:
        print("You've opened too many files")
    else:
        print("The error number is", exc.errno)
```

Gracias por vuestra atención

Iván Lago - Técnico Cisco Networking Academy ASC/ITC
PUE - ITC/ASC/CA
Área de Proyectos de Educación