2018

INTRODUCTION TO NUMERICAL ANALYSIS

Lecture 1-3: Functions and modules

Kai-Feng Chen
National Taiwan University

FUNCTIONS & MODULES

- Your life could be easier if you can build a house with lego blocks rather than sands and stones.

FUNCTIONS

- In the context of programming: a function is a named sequence of statements that performs a computation.
- An example of a function call:

```
>>> type(True)
<class 'bool'>
```

- The name of the function is **type**. The expression in **parentheses "(True)"** is the argument.
- A function "takes" an argument and "returns" a result, which is called the **return value**.

WHY FUNCTIONS?

- It is worth to divide a program into several functions:
  - It makes your program <u>easier to read</u>.
  - It makes a program <u>shorter</u> by eliminating repetitive code.
  - Dividing a long program into functions allows you to examine the parts one at a time, <u>easier to debug</u>.
  - Well-designed functions can be useful for many programs. Can be <u>reused</u> again and again.
- In python, a **module** is a file that contains a collection of related functions. It can be reused many many times.

MATH FUNCTIONS

- Python has a **math module** that provides most of the familiar mathematical functions.
- In order to use the math functions, the first step is to import the math module as:

```
>>> import math
```

The functions can be accessed by the “dot notation”:

```
>>> math.log(10)  ⇐ this is ln()
2.302585092994046
>>> math.log10(10)
1.0
```

MATH FUNCTIONS (II)

- Now you are able to calculate something much more complicated than before – e.g.

$$F(x, y) = \frac{\sin^2\left(x - \frac{\pi}{2}\right) + \cos^2\left(y + \frac{\pi}{2}\right) + e^{2(x+y)}}{(x^2 + 6)(y - 2)^3}$$

```
>>> import math
>>> F = math.sin(x - math.pi/2.)**2 +
math.cos(y + math.pi/2)**2 + math.exp((x
+y)*2)/((x**2+6)*(y-2)**3)
```

For more math functions, please see:
http://docs.python.org/3/library/math.html

THE STANDARD LIBRARY

- **Standard libraries** include many tools or functions for commonly used algorithms, data structures, and mechanisms for input and output. The math module is just one of them.
- Python has embraced a much more inclusive vision of the standard library (unlike the C/C++) — a "batteries included" philosophy.
- This is one of the special feature of python: a lot of ready-to-use tools can be included in your own coding work. If you need a quick manual, you can simply type the built-in “help” function in the python interpreter:

```
>>> import os
>>> help(os)   ⇐ will show a help page for the “os” module.
....
```

THE STANDARD LIBRARY

- help(os) may show this to you!

```
Help on module os:

NAME
    os - OS routines for NT or Posix depending on what system we're on.

MODULE REFERENCE
    https://docs.python.org/3.6/library/os

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    This exports:
      - all functions from posix or nt, e.g. unlink, stat, etc.
      - os.path is either posixpath or ntpath
      - os.name is either 'posix' or 'nt'
      - os.curdir is a string representing the current directory (always '.')
      - os.pardir is a string representing the parent directory (always '..')
      - os.sep is the (or a most common) pathname separator ('/' or '\\')
      - os.extsep is the extension separator (always '.')
```

A BRIEF TOUR

- The os module provides dozens of functions for interacting with the operating system:

```
>>> import os
>>> os.getcwd()  ⇐ get the current directory
'/Users/kfjack'
>>> os.chdir('/usr/lib')  ⇐ change to a different path
>>> os.system('ls *.a')  ⇐ run a generic system command
libcpp_kext.a      libkmodc++.a     libprofile_rt.a
libecpg.a          libl.a           libtclstub8.5.a
libecpg_compat.a   liblber.a        libtkstub8.5.a
libfl.a            libodbc.a        libwrap.a
libiodbc.a         libpgport.a      liby.a
libiodbcinst.a     libpgtypes.a     libkmod.a
libpq.a
0
```

A BRIEF TOUR (II)

- Common utility scripts often need to process command line arguments. These arguments are stored in the **sys** module's argv attribute as a list. For example:

```python
import sys
for arg in sys.argv:
    print('hello',arg)
```

helloarg.py

```
% python helloarg.py world word wood
hello helloarg.py
hello world
hello word
hello wood
%
```

Also the most direct way to terminate a code is to use **`sys.exit()`**.

A BRIEF TOUR (III)

- The **random** module provides tools for making random selections and generate random numbers:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'banana'
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.random()   ⇐ get a random float point number (uniform dist.)
0.42388613895489224
>>> random.gauss(0.,1.)   ⇐ get a random number (Gaussian dist.)
-0.7695903985126854
>>> random.randrange(1000)
267
```

We will discuss the story behind random numbers in one of later lectures.

A BRIEF TOUR (IV)

- The **datetime** module supplies classes for manipulating dates and times. Date and time arithmetic is supported.

```
>>> import datetime
>>> datetime.date.today()
datetime.date(2018, 2, 1)
>>> datetime.date.today().year
2018
>>> datetime.date.today().month
2
>>> first_day_of_ad = datetime.date(1,1,1)
>>> age = datetime.date.today() - first_day_of_ad
>>> age.days
736725
```

A BRIEF TOUR (V)

- Something cool — internet access with **urllib**:

```
>>> import urllib.request
>>> response = urllib.request.urlopen('http://
www.phys.ntu.edu.tw/')
>>> html = response.read()
>>> print(html.decode('utf-8'))

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" dir="ltr" lang="zh-tw">

<head><meta http-equiv="Content-Type" content="text/html;
charset=utf-8" /><title>
  國立臺灣大學物理學系
</title><link href="App_Themes/Theme1/Site1.css" rel="stylesheet"
type="text/css" /><link href="App_Themes/Theme1/jmenu.css"
```

COMMENTS

- Actually there are still lots of built-in standard libraries which can be very useful for your working purpose.
- Other python packages are also very useful (you will see them in some later lectures).
- Please check the official python tutorial (see the section 10 & 11): http://docs.python.org/3/tutorial/index.html
- If have no idea which package to use — **simply google your needs**. It is very easy to find a useful solution in most of cases.

INTERMISSION

- Are you able to test Euler's formula with python math module?

$$e^{i\theta} = \cos\theta + i\sin\theta$$

- How many seconds have been passed between now and the beginning of year 2000? Trick:

```
>>> import datetime
>>> datetime.datetime.now()
datetime.datetime(2018, 2, 2, 0, 3, 15, 641451)
```

DEFINE YOUR OWN FUNCTION

- The keyword **def** introduces a function definition.
- Followed by the **function name** and the **list of arguments**.
  ⇒ The rules for function names are the same as for variable.
- The body of the function must be ***indented***, similar to the if/while/for statements.
- The first statement of the function body can optionally be a string ⇒ documentation string, or **docstring**.
- It's good practice/habit to include docstrings in code, which is very useful for preparing the reference documents.

AN EXAMPLE FUNCTION

- This is a simple function that prints Fibonacci series.
- You can type it directly in your python interpreter:

```
>>> def fib(n):
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...
>>>
```

⇐ **n** is the argument of the function

⇐ To end the function, you have to enter an empty line (this is not necessary in a script).

DEFINITIONS AND USES

- Now you get a “function” object:

```
>>> print fib
<function fib at 0x1005d3b90>
>>> type(fib)
<class 'function'>
```

- To execute a function:

```
>>> fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

DEFINITIONS AND USES (II)

- Pulling together the code fragments from the previous slides, the whole program can be written as:

```
def fib(n): ⇐ ✷
    """Print a Fibonacci series up to n."""
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b

start
here
⇒ print('Print a Fibonacci series up to 1000:')
fib(1000) ⇐ call the function, jump to ✷ and back
```

printfib.py

```
% python printfib.py
Print a Fibonacci series up to 1000:
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

DEFINITIONS AND USES (III)

- The printfib.py code can be included as a module actually (suppose you put the printfib.py in your working directory):

```
>>> import printfib
Print a Fibonacci series up to 1000:
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>>
>>> printfib.fib(10000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
1597 2584 4181 6765
>>>
```

One can also put the .py file in **module searching path**, which can be set by the PYTHONPATH environment.

ARGUMENTS

- The arguments are assigned to variables called **parameters** [the “n” value in the previous fib(n) function].
- Here are another example:

```
>>> def print_twice(bruce)⇐ This function assigns the argument
...         print(bruce)       to a parameter named bruce.
...         print(bruce)
...
>>> print_twice('spam')
spam
spam
>>> print_twice(17)
17
17
```

ARGUMENTS (II)

- Basically this function works with any value that can be printed.

```
>>> print_twice(math.pi)
3.141592653589793
3.141592653589793
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

The operation has been carried out before entering the function.

⇐ The variable name dose not interfere the internal variable name (=**bruce**).

SCOPE OF VARIABLES

- When you create a variable inside a function, it is **local**, which means that it only exists inside the function.

```
def print_twice(bruce):
    print(bruce)
    print(bruce)

def cat_twice(part1,part2):
    cat = part1 + part2
    print_twice(cat)
```

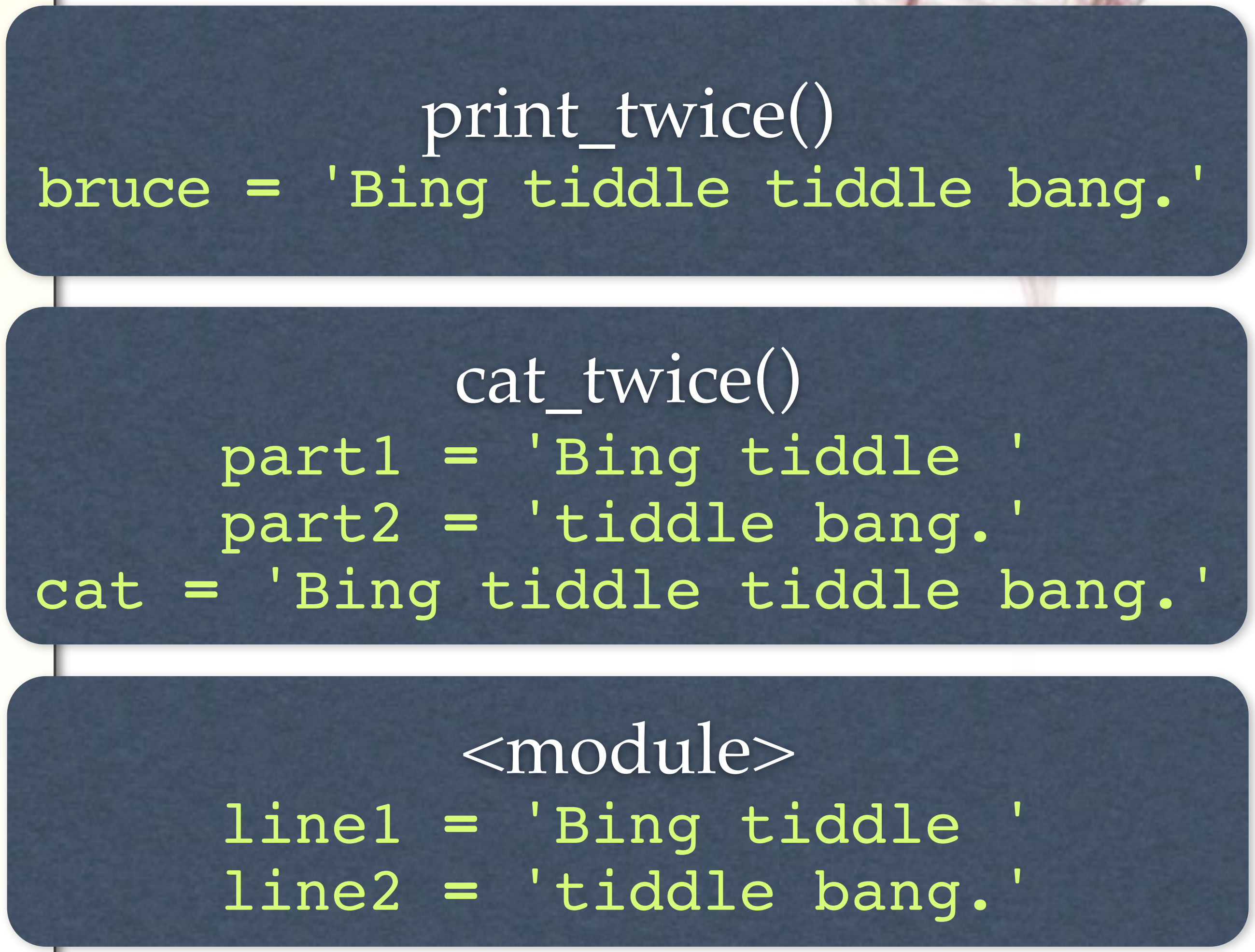
The variable bruce is local (exists) inside function print_twice; part1, part2, and cat are local to function cat_twice.

SCOPE OF VARIABLES (II)

```
def print_twice(bruce):
    print(bruce)
    print(bruce)

def cat_twice(part1,part2):
    cat = part1 + part2
    print_twice(cat)

line1 = 'Bing tiddle '
line2 = 'tiddle bang.'
cat_twice(line1, line2)
```



```
print_twice()
bruce = 'Bing tiddle tiddle bang.'

cat_twice()
part1 = 'Bing tiddle '
part2 = 'tiddle bang.'
cat = 'Bing tiddle tiddle bang.'

<module>
line1 = 'Bing tiddle '
line2 = 'tiddle bang.'
```

print_twice is called by **cat_twice**, and **cat_twice** was called by **__main__**, which is the the topmost frame. When you create a variable outside of any function, it belongs to **__main__**.

SCOPE OF VARIABLES (III)

- If you access to the variable which is not in the right scope, for example, accessing “cat” inside “print_twice”:

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
    print(cat)
```

```
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    cat_twice(line1, line2)
  File "test.py", line 8, in cat_twice
    print_twice(cat)
  File "test.py", line 4, in print_twice
    print cat
NameError: global name 'cat' is not defined
```

PYTHON SCOPE

- **A namespace is a mapping from names to objects.**
- When a name is used in a program, Python creates, changes or looks up the name in a namespace.
- A scope is a textual region of a Python program where a namespace is directly accessible.
- **Names in Python spring into existence when they are first assigned values**, and they must be assigned before used.
- Python uses the location of the assignment of a name to bind it to a particular namespace.

GLOBAL VARIABLES

- Variables defined outside of functions are belonging to **`__main__`**, or the **global variables**. One can access to the global variables within functions.
- However without the **global declaration** one cannot overwrite/modify the global variable.

```
var = 1234

def set_value():
    var = 5678 ⇐ this is actually a local variable

def show_value():
    print('var =',var)⇐ print the global var

set_value()
show_value()
```

globalvar.py

```
% python globalvar.py
var = 1234
```

GLOBAL VARIABLES (CONT.)

- One have to add the **global declaration** in order to obtain the full access to the global variable in the functions.

```
var = 1234

def set_value():
    global var ⇐ now this var is a global variable
    var = 5678

def show_value():
    print('var =',var)⇐ print the global var

set_value()
show_value()
```

globalvar.py

```
% python globalvar.py
var = 5678
```

INTERMISSION

```
def layer1(var):

    def layer2(var):
        var += 1
        print('layer2 (#1) =',var)

    var += 1
    print('layer1 (#1) =',var)
    layer2(var)
    print('layer1 (#2) =',var)

var = 1
print('global (#1) =',var)
layer1(var)
print('global (#2) =',var)
```

Try to run this code and see what are the values printed on the screen?

RETURN STATEMENT

- Some of the functions (e.g. math.sin() function), such as the math functions, have results. This is carried out by the **return** statement:

```
def factorial(n):
    """Calculate factorial of n (=n!)."""
    a = 1
    while n>1:
        a *= n
        n-=1
    return a
```

```
>>> factorial(3)
6
>>> factorial(10)
3628800
```

RETURN STATEMENT (II)

- Surely multiple return statement is allowed. This can be written in each branch with the if statement:

```
def calculate_area(x,y):
    if x<0:
        print('error: x is negative!')
        return 0
    if y<0:
        print('error: y is negative!')
        return 0

    return x*y
```

- The function **terminates** without executing any subsequent statements when it hits the return statement.

RETURN STATEMENT (III)

- When a function ends without hitting the return statement, or the function writes nothing after the return statement, it actually returns a special value called **"None"**.
- This type of function is called **void function**.

```
>>> result = print_twice('I am a void function.')
I am a void function
I am a void function
>>> print(result)
None
>>> type(result)
<class 'NoneType'>
```

MORE ON THE ARGUMENTS

- Specifying a default value for the arguments is actually allowed.
- Such function can be called with fewer arguments. e.g.:

```
def check_pin_code(pinref, retries=3):
    while True:
        pin = input('Please enter a pin code: ')
        if pin==pinref:
            return True
        else:
            print('Wrong pin code!')

        retries = retries - 1
        if retries < 0:
            print('Too many failures!')
            return False

check = check_pin_code('abcd')
```

MORE ON THE ARGUMENTS (II)

- Functions can also be called using keyword arguments of the form **argument=value**. For instance, one can call the function on the previous slide as:

```
check = check_pin_code('abcd',5)
                         Two positional arguments ⬆
check = check_pin_code('abcd',retries=5)
      1 positional argument + 1 keyword argument ⬆
check = check_pin_code(pinref='abcd',retries=5)
                             2 keyword arguments ⬆
check = check_pin_code(retries=5,pinref='abcd')
           2 keyword arguments (reversed order) ⬆
```

A MAIN FUNCTION?

- By default python does not require a **main function** to start your program. But if you want to have a main function for various reasons (e.g. to avoid immediate execution after import, to avoid unwanted confusion of global variables, etc.), it is doable.
- Here are an example:

```python
def main():
    print('Hello Main!')

if __name__ == '__main__':
    main()
```

⇐ **__name__** is a special variable contain the current name of module.

hellomain.py

```
% python hellomain.py
Hello Main!
```

RECURSION

- It is legal for one function to call another;
  it is also legal for **a function to call itself**. This is called **recursion**.
- For example:

```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```

```
>>> countdown(3)
3
2
1
Blastoff!
```

RECURSION (II)

```
>>> countdown(3)
```

```
def countdown(3):
... ...
         countdown(2)
```

```
def countdown(2):
... ...
         countdown(1)
```

```
def countdown(1):
... ...
         countdown(0)
```

```
def countdown(0):
... ...
    print('Blastoff!')
```

This is the base case and no more recursive call.

If a recursion never reaches a base case, the program will never terminate. This is known as **infinite recursion**, and it is generally not a good idea.

RECURSION (III)

- One can actually re-write the Fibonacci function with recursion:

```
def fib2(n, a=0, b=1):
    if a < n:
        print(a, end=' ')
        fib2(n, b, a+b)

print('Print a Fibonacci series up to 1000:')
fib2(1000)
```

Maybe this is slightly cooler than the original one.

INTERMISSION

- What will be the value of the variable alpha?

```
def absolute(x):
    if x>0:
        return +x
    if x<0:
        return -x

alpha = absolute(0)
```

- Let's try an infinite recursion, see if this is really run forever?

```
>>> def call_me():
...     call_me()
...
>>> call_me()
```

HANDS-ON SESSION

- **Practice 1 (a):**
  Using the random module and print 10 random numbers between 0 and 1 on your screen.
- **Practice 1 (b):**
  Generate more random numbers (e.g. 1000 numbers) and count how many random numbers are actually fall into the following intervals [0,0.25], [0.25,0.5], [0.5,0.75], and [0.75,1]. Are they all very close to 250 counts?

HANDS-ON SESSION

- **Practice 2:**
  Rewrite the following Fibonacci function. Instead of printing the series up to n, print the series up to n-th term.

```
def fib(n):
    """Print a Fibonacci series up to n."""
    a, b = 0, 1
    while a < n:
        print (a, end=' ')
        a, b = b, a+b
```

- What's the 1001th term in the series? How long (how many digits) is this number?

HANDS-ON SESSION

- **Practice 3:**
  The sine function can be expanded (approximately) as a series:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

  Write two functions sin4(x) and sin10(x), which is basically equal to the sum of first 4 and 10 terms in the series. Calculate the difference between the homemade sine functions with the standard one from the math module, for the following values of x:

  $x = \pi/8, \pi/4, \pi/2$