CS112
Introduction to Python Programming

Session 08: Functions

Shengwei Hou
Ph.D., Assistant Professor
Department of Ocean Science and Engineering
Fall 2022

SUSTech Southern University of Science and Technology

明德求是 日新自强
VIRTUE | TRUTH | ADVANCE

Contents

SUSTech Southern University of Science and Technology

- Built-in functions
- Modules
- Function definition
- The `return` statement and `void` function
- Anonymous functions: `lambda` expressions
- Scope and lifetime of variables
- Passing mutable and immutable objects
- Default parameters
- Positional and keyword arguments
- `*args` and `**kwargs`
- Command line arguments

Functions

- Functions are used when you have a block of statements that needs to be executed multiple times within the program
- This block of statements are grouped together and is given a name which can be used to invoke it from other parts of the program
- Functions also reduce the size of the program by eliminating rudimentary code
- Functions can be either built-in functions or user-defined functions

Built-in functions

- The Python interpreter has a number of functions that are built into it and are always available:

```
>>> abs(-3)
3
```

```
>>> min(1, 2, 3, 4, 5)
1
```

```
>>> max(4, 5, 6, 7, 8)
8
```

```
>>> ?divmod
Signature: divmod(x, y, /)
Docstring: Return the tuple (x//y, x%y).
Invariant: div*y + mod == x.
```

```
>>> divmod(5, 2)
(2, 1)
```

```
>>> pow(3, 2)
9
```

```
>>> len("Japan")
5
```

```
>>> ?pow
Signature: pow(base, exp, mod=None)
Docstring:
Equivalent to base**exp with 2 arguments or
base**exp % mod with 3 arguments
```

Modules 模块

- Modules in Python are reusable libraries of code having a .py extension, which implements a group of methods and statements
- To use a module in your program, import the module using the `import` statement:

```
>>> import math
>>> math.ceil(5.4)
6
>>> math.sqrt(4)
2.0
>>> math.cos(1)
0.5403023058681398
>>> math.factorial(6)
720
>>> math.pow(2, 3)
8.0
```

Modules

- The built-in function `dir()` returns a sorted list of comma separated strings containing the names of functions, classes and variables as defined in the module

```
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__',
'__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2',
'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

Modules

- The `help()` function invokes the built-in help system:

```
>>> help(math.pow)

Help on built-in function pow in module math:

pow(x, y, /)
    Return x**y (x to the power of y).
```

- The `random` module generates random numbers:

```
>>> import random
>>> random.random()
0.3377069844588696
>>> random.randint(5,10)
9
```

Function definition

- User-defined functions are reusable code blocks created by users to perform some specific task in the program:

```
def function_name(parameter_1,…, parameter_n):
    statement(s)
```

- In Python, a function definition consists of the **`def`** keyword, followed by:
  - The function name: use letters, numbers, or an underscore, but the name cannot start with a number
  - A list of parameters enclosed in "`()`" and separated by "`,`". Some functions may not have parameters
  - A colon at the end of the function header
  - Block of statements that define the body of the function start and they must have the same indentation level

Function definition

- The first statement among the block of statements can optionally be a documentation string or docstring:

```
""" This is single line docstring """
```

OR

```
""" This is
multiline
docstring """
```

Function definition

- The syntax for function call or calling function is:

```
function_name(arg_1, arg_2,…,arg_n)
```

- Arguments are the actual value that is passed into the calling function. There must be a one-to-one correspondence between the parameters in the function definition and the actual arguments of the calling function

- A function should be defined before it is called and the block of statements in the function definition are executed only after calling the function

Function definition

- If the Python interpreter is running the source program as a stand-alone main program, it sets the special built-in `__name__` variable to have a string value `"__main__"`:

```
    def function_with_no_argument():
        print("This is a function with NO Argument")

    def function_with_one_argument(message):
        print(f"This is a function with {message}")

    def main():
        function_with_no_argument()
        function_with_one_argument("One Argument")

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

/ main()

Write your first function

- Write a function to calculate the area of a square with side length a

Write your first function

- Write a function to print the area of a square with side length *a*

```
def area_square(a):
    area = a * a
    print(f"The area is {area}")
```

```
>>> area_square(1)
The area is 1
>>> area_square(a = 3)
The area is 9
```

```
>>> b = 2
>>> area_square(b)
The area is 4
>>> area_square(a = b)
The area is 4
```

The `return` statement and `void` function

- Most of the times you may want the function to perform its specified task to calculate a value and return the value to the calling function. This can be achieved using the optional `return` statement in the function definition

```
return [expression_list]
```

- The `return` statement terminates the function definition and returns to the calling function with an optional value

- It is possible to define functions without a `return` statement. Functions like this are called void functions, and they return `None`

The `return` statement and `void` function

- If you want to return a value using the `return` statement from the function definition, then you have to assign the result of the function to a variable

- A function can return only a single value, but that value can be a list or tuple

- When returning multiple values, separating them by a comma will by default construct a tuple by Python

The `return` statement and `void` function

- When calling function receives a tuple from the function, it is common to assign the result to multiple variables by specifying the same number of variables on the left-hand side of the assignment. This is called tuple unpacking:

```
def buy_apple():
    price = input("What is the price of an apple?")
    amount = input("How many do you want?")
    return price, amount, float(price) * float(amount)

if __name__ == "__main__":
    price, amount, cost = buy_apple()
    print(f"You need to pay {cost} Yuan for {amount} apple(s)")
```

Anonymous functions: `lambda` expressions

- A `lambda` expression is an in-line function that can be generated on the fly to accomplish small tasks, often where a function name is needed as input to another function:

```
lambda arg1, arg2, ... : output
```

```
>>> g = lambda a, b: 3*a + b**2
>>> g(2,3)
15
```

Scope and lifetime of variables

- Python programs have two scopes: global and local
- Global variable is accessible and modifiable throughout the program. A variable that is defined inside a function definition is a local variable
- The local variable is created and destroyed every time the function is executed, and it cannot be accessed by any code outside the function definition
- Global variables are accessible from inside a function, as long as you have not defined a local variable with the same name
- A local variable can have the same name as a global variable, but they are totally different so changing the value of the local variable has no effect on the global variable. Only the local variable has meaning inside the function in which it is defined

```
>>> a  =  1
>>> def test_a(x):
        a = 2
        a += x
        return a, x
```

```
>>> local_a, global_a = test_a(a)
>>> print(local_a, global_a)
3 1
```

Scope and lifetime of variables


```
variable = 5
def outer_function():
    variable = 60
    def inner_function():
        variable = 100
        print(f"Local variable of value {variable}")
    inner_function()
    print(f"Local variable of value {variable}")
outer_function()
print(f"Global variable of value {variable}")
```

Output:

```
Local variable of value 100
Local variable of value 60
Global variable of value 5
```

Scope and lifetime of variables


```
variable = 5
def outer_function():
    variable = 60
    def inner_function():
        variable = 100
        print(f"Local variable of value {variable}")
    inner_function()
    print(f"Local variable of value {variable}")
outer_function()
print(f"Global variable of value {variable}")
```

- It is not recommended to access global variables from inside the definition of the function
- If there is a need by function to access an external value, then it should be passed as a parameter to that function

Scope and lifetime of variables

- A function definition can be nested within another function definition
- The inner function can use the arguments and variables of the outer function, while the outer function cannot use the arguments and variables of the inner function
- The inner function definition can be invoked by calling it from within the outer function definition:

```
def add_cubes(a, b):
    def cube_surface_area(x):
        return 6 * pow(x, 2)
    return cube_surface_area(a) + cube_surface_area(b)

result = add_cubes(2, 3)
print(f"The total surface area is {result}")
```

```
The total surface area is 78
```

Passing mutatable and immutable objects

- Numbers, strings and tuples are immutable; changing them within a function creates new objects with the same name inside of the function, but the old objects remain unchanged
- Changes to immutable arguments of a function within the function do not affect their values in the calling program

- Lists and arrays are mutable; those elements that are changed inside the function are also changed in the calling function
- Changes to mutable arguments of a function within the function are reflected in the values of the same list and array elements in the calling function

Passing mutatable and immutable objects

String

```
def test(s):
    s = "Hello again"
    return s

s = "Hello"
s1 = test(s)
print(f"s = {s}")
print(f"s1 = {s1}")
print(f"s = {s}")
```

```
s = Hello
s1 = Hello again
s = Hello
```

Number

```
def test(n):
    n = 100
    return n

n = 30
n1 = test(n)
print(f"n = {n}")
print(f"n1 = {n1}")
print(f"n = {n}")
```

```
n = 30
n1 = 100
n = 30
```

Tuple

```
def test(t):
    t = (1, 2)
    return t

t = (3, 4)
t1 = test(t)
print(f"t = {t}")
print(f"t1 = {t1}")
print(f"t = {t}")
```

```
t = (3, 4)
t1 = (1, 2)
t = (3, 4)
```

Passing mutatable and immutable objects

SUSTech Southern University of Science and Technology

List

```
def test(l):
    l[-1] = "end"
    return l

l = [1, 2]
print(f"l = {l}")
l1 = test(l)
print(f"l1 = {l1}")
print(f"l = {l}")
```

```
l = [1, 2]
l1 = [1, 'end']
l = [1, 'end']
```

Numpy array

```
import numpy as np
def test(a):
    a[0] = 100
    return a

a = np.zeros(2)
print(f"a = {a}")
a1 = test(a)
print(f"a1 = {a1}")
print(f"a = {a}")
```

```
a = [0.   0.]
a1 = [100.   0.]
a = [100.   0.]
```

Default parameters

- In some situations, it might be useful to set a default value to the parameters of the function definition. Each default parameter has a default value as part of its function definition

- Any calling function must provide arguments for all required parameters in the function definition but can omit arguments for default parameters. If no argument is sent for that parameter, the default value is used

Default parameters

- Usually, the default parameters are defined at the end of the parameter list:

```
def work_area(prompt, domain="Bioinformatics"):
    print(f"{prompt} {domain}")
work_area("Sam works in")
work_area("Alice has interest in", "Genomics")
```

- Output:

```
Sam works in Bioinformatics
Alice has interest in Genomics
```

Positional and keyword arguments

- Generally, whenever you call a function with some values as its arguments, these values get assigned to the parameters in the function definition according to their position
- You can also explicitly specify the keyword argument name along with its value in the form `kwarg = value`. The normal arguments without keywords are called positional arguments
- In the calling function, keyword arguments must follow positional arguments
- All the keyword arguments passed must match one of the parameters in the function definition and their order is not important
- No parameter in the function definition may receive a value more than once

Positional and keyword arguments


```
def area_rectangle(a, b = 6):
    area = a * b
    print(f'Side a is {a}, side b is {b}')
    print(f'The area is {area}')
```

```
>>> area_rectangle(4)
Side a is 4, side b is 6
The area is 24
```

```
>>> area_rectangle(a = 4)
Side a is 4, side b is 6
The area is 24
```

```
>>> area_rectangle(4, 5)
Side a is 4, side b is 5
The area is 20
```

```
>>> area_rectangle(5, 4)
Side a is 5, side b is 4
The area is 20
```

```
>>> area_rectangle(a = 4, b = 5)
Side a is 4, side b is 5
The area is 20
```

```
>>> area_rectangle(b = 5, a = 4)
Side a is 4, side b is 5
The area is 20
```

Positional and keyword arguments


```
def area_rectangle(a, b = 6):
    area = a * b
    print(f'Side a is {a}, side b is {b}')
    print(f'The area is {area}')
```

Invalid function calls:

```
>>> area_rectangle()
TypeError: area_rectangle() missing 1 required positional argument: 'a'

>>> area_rectangle(a = 5, 4)
SyntaxError: positional argument follows keyword argument

>>> area_rectangle(5, a = 5)
TypeError: area_rectangle() got multiple values for argument 'a'

>>> area_rectangle(5, c = 4)
TypeError: area_rectangle() got an unexpected keyword argument 'c'
```

*args and **kwargs

- `*args` and `**kwargs` allow you to pass a variable number of arguments to the function definition. It is very useful when user does not know in advance about how many arguments will be passed to the function definition

- `*args` as parameter in function definition allows you to pass a non-keyworded & variable length tuple argument list to the function definition

- `**kwargs` as parameter in function definition allows you to pass keyworded & variable length dictionary argument list to the function definition

*args and **kwargs

- `*args` must come after all the positional parameters and `**kwargs` must come right at the end

- The single asterisk (*) and double asterisk (**) are the important elements here and the words `args` and `kwargs` are used only by convention. The Python does not enforce those words and the user is free to choose any words of their choice

*args and **kwargs


```
def fruit_shop(*args, **kwargs):
    print(args, kwargs)
    fruits = ", ".join(args)
    print(f"Do you have: {fruits}?")
    print(f"Yes, the price is:")
    for kw in kwargs:
        print(f"{kw}: ${kwargs[kw]}/each")
```

```
>>> fruit_shop("banana", "orange", banana = 3, orange = 7)
('banana', 'orange') {'banana': 3, 'orange': 7}
Do you have: banana, orange?
Yes, the price is:
banana: $3/each
orange: $7/each
```

Command line arguments

- A Python program can accept any number of arguments from the command line
- Command line arguments is a methodology in which user will give inputs to the program through the console using commands
- You need to import `sys` module to access command line arguments. All the command line arguments in Python can be printed as a list of string by executing `sys.argv`

Command line arguments

example.py

```
import sys
def main():
    for arg in sys.argv:
        print(arg)
if __name__ == "__main__":
    main()
```

In your terminal:

```
$ python example.py first_arg 100 10.0
main.py
first_arg
100
10.0
```

Exercises

Exercise 1

Write a program including a function to find and print the area of trapezium using the formula $area = (1/2) * (a + b) * h$.

Here, a and b are the two bases of trapezium and h is the height obtained from keyboard input.

Exercise 2

Write a function that outputs (x,y)-coordinates of n points evenly distributed around a circle of radius r centered at the point (x_0,y_0) where n is obtained from keyboard input, while (x_0,y_0) and r are as follows: $(x_0,y_0) = (2,3)$, $r = 5$

Exercise 3

A three-digit number is called an Armstrong number if the sum of cube of its digits is equal to the number itself, e.g., 407.

Write a program to check if a three-digit number is an Armstrong number.

Exercise 4

Write a function that could accept a variable number of values and calculate their product.

Exercise 5

Write a function that could return the max of three numbers x, y and z.

Note: You can not use the `max()` directly.

Exercise 6

Write a Python function that accepts a string and calculate the number of upper case letters and lower case letters.

Exercise 7

Write a Python function to check whether a number is perfect or not.

A **perfect number** is a positive integer that is equal to the sum of its proper positive divisors (excluding itself).

For example:

- The first perfect number is 6, because 1, 2, and 3 are its proper positive divisors, and 1 + 2 + 3 = 6.
- The next perfect number is 28 = 1 + 2 + 4 + 7 + 14. This is followed by the perfect numbers 496 and 8128.