Python: Functions & Methods

1405 Instructor: Ruiqing (Sam) Cao

Basic Types of Functions

- **Built-in functions**: These are Python-provided functions that are globally available without requiring an import

Examples: `len()` `print()`

- **Imported functions**: These are functions defined in an external library which you can use after importing the library

Examples: `math.sqrt()` `requests.get()`

- **User-defined functions**: These are functions defined by yourself to perform certain tasks

Example: `def function_name(args):...`

Define a Function

- **Functions** are defined using the `def` keyword, has a *name*, a *body*, *(optional) arguments*, and *(optional) return statement*

- Functions without a return statement return None

```
def function_name(args):
    ...
    return(result)
```

or

```
def function_name(args):
    ...
    return result
```

Call a Function

- To call a function, simply reference its name followed by parentheses enclosing *optional parameters* passed as arguments
- The *value of the function* equal the returned result from running the code block in the function’s body

```
def function_name(args):
    ...
    return(result)
function_value = function_name(args=params)
```

Pass Positional Arguments to a Function

- *Positional* arguments

```
def function_name(id,name,action):
    ...
function_value = function_name(p0,p1,p2)
```

id ← p0 name ← p1 action ← p2

- If a parameter is passed to the function *with no keyword specified in the n-th position*, then it automatically fills *the n-th argument* in the definition of the function

Pass Keyword Arguments to a Function

- *Keyword* arguments

```
def function_name(id,name,action):
    ...
function_value = function_name(name=p1,action=p2,id=p0)
```

name ← p1   action ← p2   id ← p0

- If a parameter is passed to the function *with a keyword*, then it automatically fills the *argument corresponding to that keyword* regardless of its position in the argument

Default Arguments & Overwriting Them

- Arguments with *default* parameter values can be *overwritten*

```
def function_name(id,name='',action=None):
    ...
function_value = function_name(id=123,name='abc')
```

```
action ← None id ← 123 name ← 'abc'
```

- When a function is defined, default values can be specified for some or all arguments (optional)

➢The function runs with the default value if no parameter is passed

➢If a parameter is passed for an argument, it overwrites the default value

Exercises: Function Basics

1. Write a function that takes two parameters `li` (list) and `tf` (bool)
2. The function takes a list of numbers in `li`, calculates and returns the *average* of its elements if `tf` is True, calculates and returns the *sum* of its elements if `tf` is False, and returns an error message if `li` is empty or contains non-numeric elements (hint: just throw an exception)
3. Call the function using positional arguments, with parameters `[1,2,3]` for `li` and `True` for `tf`
4. Call the function using keyword arguments, with parameters `[1,2,3]` for `li` and `True` for `tf`

Global vs. Local Variables

- It’s important to distinguish between variables *inside* a function (local variable) and variables *outside* of it (global variable)

- **Local variables**: declared inside a function and only accessible within that function

- **Global variables**: declared outside all functions and accessible throughout the program, including inside functions

Modify Local Variables Inside Function

Local variables (within a function)

- Either passed as an argument or declared within the function
- Do not exist or retain value outside the function, or once the program finishes executing the function

Global vs. Local Variables: Arguments

- When a variable is passed as an argument into a function, it's considered a local variable within the function, but its behavior differs depending on data type

➢Access an *immutable* type by copying its entire value

→ Immutable types: `int`, `float`, `str`, `bool`…

➢Access a *mutable* type by pointing to its location in the memory

→ Mutable types: `list`, `dict`…

Global vs. Local Variables: Naming

When a local variable and a global variable have the *exact same name*, the local variable takes precedent

- The code inside the function that refers to that name refers to the <u>local variable</u> (not the global variable) by default

The best practice is to simply *AVOID USING THE SAME NAMES* for global and local variables

Exercises: Local Variables (Practice Quiz)

- What are the outputs of these Python code snippets below?

```python
def change(i):
    i=i+1
    return i
i=0
change(1)
print(i)
```

```python
def foo(k):
    k[0] = 1
q = [0]
foo(q)
print(q)
print(k)
```

← Can we print k here?

Exercises: Local Variables (Practice Quiz)

- What are the outputs of these Python code snippets below?

```
def change(i):
    i=i+1
    return i
i=0
change(1)
print(i)
```

2 - anyone?

1 - anyone?

```
def foo(k):
    k[0] = 1
q = [0]
foo(q)
print(q)
print(k)
```

`[0]` - anyone?

*← **Can we print k here?***

Exercises: Local Variables (Practice Quiz)

- What are the outputs of these Python code snippets below?

```
def change(i):
    i=i+1
    return i
i=0
change(1)
print(i)
```

2 - anyone?

1 - anyone?

```
def foo(k):
    k[0] = 1
q = [0]
foo(q)
print(q)
print(k)
```

`[0]` - anyone?

← ***Can we print k here?***

Exercises: Local Variables (Practice Quiz)

- What are the outputs of these Python code snippets below?

Which i is this? Is it local or global? What is its value?

```
def change(i):
    i=i+1
    return i
i=0
change(1)
print(i)
```

Which i is this? Is it local or global? What is its value?

```
def foo(k):
    k[0] = 1
q = [0]
foo(q)
print(q)
print(k)
```

What is the relationship between k and q?

← *Can we print k here? No.*

Modify Global Variables Inside Function

Global variables

- Are declared outside all functions
- Can be accessed inside a function
- By default, cannot be modified inside a function
- To modify the global variable `var` inside a function, declare "`global var`" inside that function; Then `var` changes and retains value outside the function if the function modifies it

Exercises: Global Variables (Practice Quiz)

- What are the outputs of these Python code snippets below?

```
def foo():
    return total + 1
total = 0
print(foo())
```

```
y, z = 1, 2
def f():
    global tt
    tt = y+z
f()
print(tt)
```

Exercises: Global Variables (Practice Quiz)

- What are the outputs of these Python code snippets below?

```python
def foo():
    return total + 1
total = 0
print(foo())
```

Someone says: total should not be accessed inside foo() !

```python
y, z = 1, 2
def f():
    global tt
    tt = y+z
f()
print(tt)
```

Someone says: tt should not exist outside f() !

Exercises: Global Variables (Practice Quiz)

- What are the outputs of these Python code snippets below?

```
def foo():
    return total + 1
total = 0
print(foo())
```

```
y, z = 1, 2
def f():
    global tt
    tt = y+z
f()
print(tt)
```

Someone says: total should not be accessed inside foo() !

Someone says: tt should not exist outside f() !

Exercises: Global Variables (Practice Quiz)

- What are the outputs of these Python code snippets below?

```
def foo():
    return total + 1
total = 0
print(foo())
```

Global variable total can be accessed inside a function

```
y, z = 1, 2
def f():
    global tt
    tt = y+z
f()
print(tt)
```

Global variable tt can be modified inside a function after it is declared explicitly

Writing Good User-Defined Functions

To avoid errors with user-defined functions, follow these best practices:

- **Avoid Variable Name Conflicts:** Never use the same name for different variables, especially when one is global and the other is local
- **Keep Functions Independent:** Functions should be self-contained and reusable across various programs without modifications
- **Design Arguments and Returns Carefully:** Pass required data as arguments instead of relying on global variables. Always return results rather than modifying global variables directly
- **Treat Global Variables as Constants:** Access global variables *without changing them* inside a function. If modification is necessary, declare them explicitly with `global [varname]` inside the function

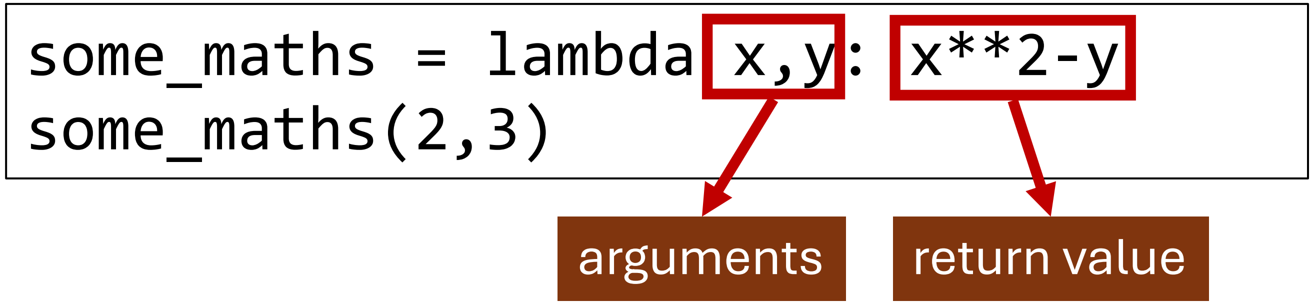
Lambda Function

- **Lambda functions**: these functions *do not need to have names*, and they are *one-line functions* defined using the `lambda` keyword

User-defined function:

```
def some_maths(x,y):
    return x**2-y
some_maths(2,3)
```

Equivalent lambda function:



```
some_maths = lambda x,y: x**2-y
some_maths(2,3)
```

Lambda Function

- **Lambda functions**: these functions *do not need to have names*, and they are *one-line functions* defined using the `lambda` keyword

```
some_maths = lambda x,y: x**2-y
some_maths(2,3)
```

- Lambda function is useful to data professionals, because it allows you to write one-line code *applying the same function to all rows* in a data table without iterating through it using a `for` loop

```
df['new_var'] = df['var'].apply(lambda x: x**2-1)
```

→ For instance, the line above generates a new column `'new_var'` equal to x^2-1 for all x in the column `'var'` of the Pandas DataFrame named `df`

Exercises: Lambda Function

1. Write a regular function that takes in a list (or a string) as an argument, reverses the list (or string), and returns it.
2. Calls the function with a user-provided list (or string) and prints the reversed list.
3. Write a lambda function that performs the same operation as the regular function.
4. Create a list of strings, e.g., ['banana', 'potato', 'tomato'] and produce a new list where each element is a reversed string from the original list [Hint: use `map(function,list)`]

Functions & Methods

- You have encountered both *functions* and *methods* in this course, and will continue to see them if you write Python code

→ They can be very different despite looking similar, and it is important to *understand their differences*

	Function	Method
Definition	Reusable block of code	A function defined inside a *class*
Usage	Perform an *independent* task, not tied to any object or class	*Operates on an object itself* and tied to a class and its instances
Syntax	`function_name(arguments)`	`obj.method_name(arguments)`

Functions vs. Methods Comparison

Can perform similar tasks, for instance:

- Sort a list:

Function

```
li_new=sorted(li)
```

Method

```
li.sort()
```

- Combine lists:

Function

```
func=lambda x,y: x+y
li_new=func(l1,l2)
```

Method

```
li1.extend(li2)
```

Some methods in NumPy and Pandas have `inplace` options (but not functions), e.g., `df.dropna()` directly modifies `df` if `inplace=True` is passed as an argument, but the same method returns a new object by default `inplace=False`

Functions & Methods Best Practices

Aim to produce clean, readable, and efficient code

- Functions
  - Create functions for *repeatedly executed tasks*
  - Use *descriptive names* to define functions
  - *Document the required arguments* clearly using comments

- Methods
  - Use *built-in methods* (e.g., `list.append()`, `str.lower()`)

Modules

Modules

- *Modules*: self-contained packages containing functions, classes, and variables that can be imported and re-used

Think: Lego pieces that can be re-used to build different toys

➢ *Built-In modules*: provided by Python’s standard library

e.g., `math`, `os`, `subprocess`

➢ *External modules*: must be *installed* from external sources

e.g., `request`, `numpy`, `pandas`

Install External Modules

To import a module, it must already exist in your Python environment (e.g., the virtual environment for your project)

- Built-in modules do not need to be installed, but *<u>external modules must be installed</u>* first

To install an external module, you can do one of the following:

- Run `pip install module` in command line (PC or Mac)
- Run `%pip install module` in a Jupyter Notebook code block

Import Modules

To use an existing `module`, you must *import it* `import module`

- Call a function `func` in an imported module: `module.func()`

Or *import func from the module* `from module import func`

- Call the imported `func` directly: `func()`

Or import a `module` and *rename* it `import module as md`

- Call `func` in the imported and renamed module: `md.func()`