

Topics

- **Functions**
- **Positional argument**
- **Name space**
- **Advanced functions**
- **Error handling**

Functions

- Functions are an extremely important programming concept for structuring your code and avoiding repetitions

1. Keyword **def** marks the start of function header.
2. A function name to **uniquely identify** it. Function naming follows the same rules of writing identifiers in Python.
3. Arguments through which we pass values to a function. They are optional.
4. A colon (:) to mark the end of function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have same indentation level (**usually 4 spaces**).
7. An optional **return statement** to return a value from the function.

```
def Hello():  
    print('hello')
```

Global variable : 'global' keyword

- Share a global variable across all Python Modules

```
c = 1 # global variable
def add():
    print(c)
```

```
c = 1 # global variable
def add():
    c = c + 2 # or c = 4
    print(c)
```

```
c = 0 # global variable
def add():
    global c
    c = c + 2
    print("Inside add():", c)
```

```
add()
print("In main:", c)
```

```
c = 1 # global variable
def add():
    c = 4
    print(c)
```

Global variable : 'global' keyword

- Each function defines its own namespace
- If you define a variable called x in a main program and another variable called x in a function, they refer to different things
- The main part of a program defines the *global* namespace

```
x = 4
y = 6
def adding():
    return x+y
>>> adding()

def adding():
    x = 10; y = 15
    return x+y
>>> adding()
>>> print(x, y)
```

Name space

- The main part of a program defines the *global* namespace; thus, the variables in that namespace are *global variables*
- You can get the value of a global variable from within a function:

```
animal = 'fruitbat'
```

```
def print_global():  
    print('inside print_global:', animal)
```

```
def change_and_print_global():  
    print('inside change_and_print_global:', animal)  
    animal = 'wombat'  
    print('after the change:', animal)
```

If you try to get the value of the global variable and change it within the function, you get an error:

Name space

- The `change_local()` function also has a variable named `animal`, but that's in its local namespace
- To access the global variable rather than the local one within a function, you need to be explicit and use the `global` keyword
- Local variable goes away (removed from the memory) after the function completes (no return value)

```
animal = 'fruitbat'
```

```
def change_local():  
    animal = 'wombat'  
    print('inside change_local:', animal, id(animal))
```

```
>>> change_local()  
1651386275800  
>>> id(animal)  
1651386174128
```

Name space

- Python provides two functions to access the contents of your namespaces:
 - ✓ `locals()` returns a dictionary of the contents of the local namespace.
 - ✓ `globals()` returns a dictionary of the contents of the global namespace.

```
animal = 'fruitbat'
def change_local():
    a = 4; b = 6; c = 'hello'
    animal = 'wombat' # local variable
    print('locals:', locals())
    tm = locals()
    return tm
```

```
>>> l_var = change_local()
>>> g_var = globals()
```

Use of '_' and '__'

- Names that begin and end with two underscores (__) are reserved
- It seemed unlikely to be selected by application developers

```
def amazing():  
    """This is the amazing function.  
    Want to see it again?"""  
    print('This function is named:', amazing.__name__)  
    print('And its docstring is:', amazing.__doc__)
```

```
>>> amazing()  
This function is named: amazing  
And its docstring is: This is the amazing function.  
    Want to see it again?
```


Functions can be argument

- In Python, parentheses () mean *call this function*
- With no parentheses, Python just treats the function like any other object.

```
def my_add(arg1, arg2):  
    return arg1+arg2
```

```
def my_mul(arg1, arg2):  
    return arg1*arg2
```

```
def run_this_functions(some_fun, arg1, arg2):  
    some_fun(arg1, arg2)
```

```
>>> type(my_add)  
>>> run_this_functions(my_add, 5, 6)  
>>> run_this_functions(my_mul, 5, 6)
```

Inner function, Closure

- A function is an instance of the Object type.
- You can store the function in a variable.
- You can pass the function as a parameter to another function.
- You can return the function from a function.
- You can store them in data structures such as hash tables, lists, ...

Inner functions

A function which is defined inside another function is known as inner function or nested function. Nested functions are able to access variables of the enclosing scope. Inner functions are used so that they can be protected from everything happening outside the function. This process is also known as Encapsulation.

Closures

A Closure is a function object that remembers values in enclosing scopes even if they are not present in memory.

Inner functions

- You can define a function within another function
- An inner function can be useful when performing some complex task more than once within another function

```
def outer(a, b):  
    def inner(c, d):  
        return c + d  
    return inner(a, b)
```

```
>>> outer(4, 7)  
11  
>>> inner(4,5) # ??
```

```
def knights(saying):  
    def inner(quote):  
        return "We are the knights who say: '%s'" % quote  
    return inner(saying)
```

```
>>> knights('Ni!')  
"We are the knights who say: 'Ni!'"
```

Closure

- This is a function that is dynamically generated by another function
- Remember the values of variables that were created outside the function

```
def knights2(saying):  
    def inner2():  
        return "My country is: '%s'" % saying  
    return inner2
```

```
>>> a = knights2('Kazakhstan')  
>>> type(a)  
<function __main__.knights2.<locals>.inner2>  
>>> b = knights2('Korea')  
>>> a()  
"My country: 'Kazakhstan'"  
>>> b()  
"My country: 'Korea'"
```

Inner function / closure

```
nu = {'salary': 100, 'w_hours': 40, 'country': 'Kazakhstan'}  
ku = {'salary': 110, 'w_hours': 45, 'country': 'Korea'}
```

```
print_info('Min-ho lee', 'Korea University')
```

Output

```
Name: Min-ho lee  
Affiliation: Korea University  
Country: Korea  
working hours: 45  
Salary: 110
```

Inner function / Closure

```
nu = {'salary': 100, 'w_hours': 40, 'country': 'Kazakhstan'}  
ku = {'salary': 110, 'w_hours': 45, 'country': 'Korea'}
```

```
def print_info(name='Unkown', base_uni = 'Nazarbayev'):
```

```
    if base_uni == 'Nazarbayev University':  
        print('Name:', name, '\nAffiliation:', base_uni, '\nCountry:', nu.get('country'))  
        print('working hours:', nu.get('w_hours'), '\nSalary:', nu.get('salary'))
```

```
    if base_uni == 'Korea University':  
        print('Name:', name, '\nAffiliation:', base_uni, '\nCountry:', KU.get('country'))  
        print('working hours:', KU.get('w_hours'), '\nSalary:', KU.get('salary'))
```

```
def outer_intro(name='Unkown', base_uni = 'Nazarbayev'):
```

```
    def inner_intro(name='Unkown'):
```

```
        if base_uni == 'Nazarbayev':  
            Uni = {'salary': 100, 'w_hours': 40, 'country': 'Kazakhstan'}
```

```
        if base_uni == 'Korea':  
            Uni = {'salary': 110, 'w_hours': 45, 'country': 'Korea'}
```

```
        print('Name:', name, '\nAffiliation:', base_uni, '\nCountry:', Uni.get('country'))  
        print('working hours:', Uni.get('w_hours'), '\nSalary:', Uni.get('salary'))
```

```
    return inner_intro(name)
```

Name	Type	Size	Value
ku	dict	3	{'salary':110, 'w_hours':45, 'country': 'Korea'}
nu	dict	3	{'salary':100, 'w_hours':40, 'country': 'Kazakhstan'}
print_info	function	1	function object

Output

```
Name: Min-ho lee  
Affiliation: Korea  
University  
Country: Kazakhstan  
working hours: 40  
Salary: 100
```

Name	Type	Size	Value
outer_intro	function	1	function object

Decorator

- Sometimes, you want to modify an existing function without changing its source code
- Example is adding a debugging statement to see what arguments were passed in.

```
def add_ints(a, b):  
    return a + b
```

```
def add_ints(a, b):  
    print('Running....')  
    print('Positional....')  
    print('Positional....')  
    print('Positional....')  
    return a + b
```

```
>>> add_ints(4,5)  
9
```

```
>>> add_ints(4,5)  
Running function: add_ints  
Positional arguments: (4, 5)  
Keyword arguments: {}  
Result: 9
```

Decorator

- A *decorator* is a function that takes one function as input and returns another function
 - `*args` and `**kwargs`
 - Inner functions
 - Functions as arguments

```
def document_it(func):  
    def new_function(*args, **kwargs):  
        print('Running function:', func.__name__)  
        print('Positional arguments:', args)  
        print('Keyword arguments:', kwargs)  
        result = func(*args, **kwargs)  
        print('Result:', result)  
        return result  
    return new_function
```

```
>>> better_add_ints = document_it(add_ints)  
>>> better_add_ints(4, 5)
```


Decorator

```
def document_it(func):
    def new_function(*args, **kwargs):
        print('Running function:', func.__name__)
        print('Positional arguments:', args)
        print('Keyword arguments:', kwargs)
        result = func(*args, **kwargs)
        print('Result:', result)
    return result
return new_function
```

decorator

```
def add_ints(a, b):
    return a + b
```

```
def mul_ints(a, b):
    return a * b
```

```
@document_it
def add_ints(a, b):
    return a + b
```

```
@document_it
def mul_ints(a, b):
    return a + b
```

```
>>> add_ints(4, 5)
```

```
>>> mul_ints(4, 5)
```

Decorator

▪ Tracking the changes of arguments

```
def print_it(func):  
    def new_function(*args, **kwargs):  
        result = func(*args, **kwargs)  
        print('Result:', args)  
        return result  
    return new_function  
  
@print_it  
def add_ints(a, b):  
    return a + b  
  
a = 3; b = 6  
for i in range(1, 10):  
    a, b = add_ints(a, b), add_ints(b, a)/2
```

Anonymous Functions: lambda() Function

- An anonymous function expressed as a single statement. You can use it instead of a normal tiny function.
- The lambda takes one argument. Everything between the colon and the terminating parenthesis is the definition of the function.
- Temporal function/remove, no **return** keyword

```
stairs = ['thud', 'meow', 'thud', 'hiss']
```

```
def edit_story(words, func):
```

```
    for word in words:  
        print(func(word))
```

```
def enliven(word): # give that prose more punch  
    return word.capitalize() + '!'
```

```
>>> edit_story(stairs, enliven)
```

```
>>> edit_story(stairs, lambda word: word.capitalize() + '!')
```

Anonymous Functions: lambda() Function

- Often, using real functions such as `enliven()` is much clearer than using lambdas
- Lambdas are useful for cases in which you would otherwise need to define many tiny functions and remember what you called them all

```
g = lambda x: x**2
f = lambda x, y: x + y

def inc(n):
    return lambda x: x + n
```

```
>>> g(8)
>>> f(4, 4)
>>> f = inc(2)
>>> g = inc(4)
>>> f(12)
>>> g(12)
>>> inc(2)((12))
```

```
n = 10
def inc2(x, *arg):
    print(len(arg))
    if not arg:
        global n
        return x + n
    else:
        n = arg[0]
        print(n)
        return x + n
```

```
>>> inc2(3)
>>> inc2(3, 5)
```

Error handling : try, except

- Python uses *exceptions*: code that is executed when an associated error occurs.
- The finally block consists of statements which should be processed regardless of an exception occurring in the try block or not.

try :

#statements in try block

except :

#executed when error in try block

Try :

#statements in try block

except :

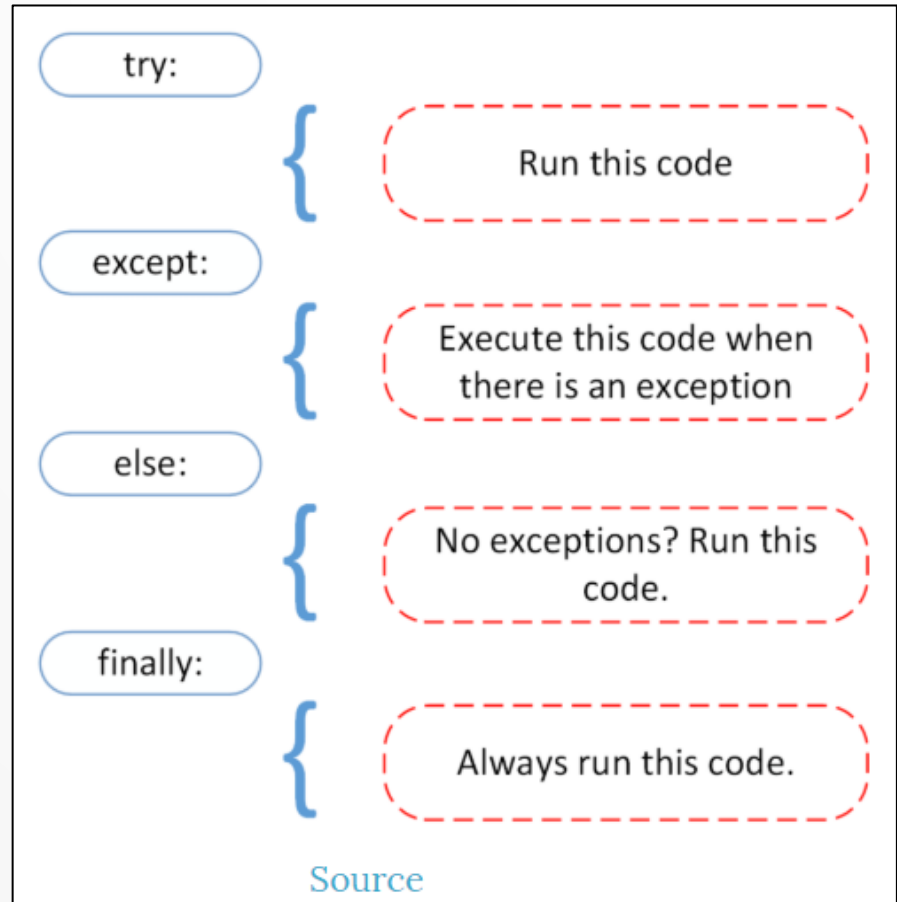
#executed when error in try block

else :

#executed if try block is error-free

finally :

#executed irrespective of exception occurred or not



Error handling : try, except

try:

```
print("try block")
x=int(input('Enter a number: '))
y=int(input('Enter another number: '))
z=x/y
```

except ZeroDivisionError:

```
print("except ZeroDivisionError block")
print("Division by 0 not accepted")
```

else:

```
print("else block")
print("Division = ", z)
```

finally:

```
print("finally block")
x=0
y=0
print ("Out of try, except, else and finally blocks." )
```

Type of errors

- **IndexError** is thrown when trying to access an item at an invalid index.
- **ModuleNotFoundError** is thrown when a module could not be found.
- **KeyError** is thrown when a key is not found.
- **ImportError** is thrown when a specified function can not be found.
- **TypeError** is thrown when an operation or function is applied to an object of an inappropriate type.
- **ValueError** is thrown when a function's argument is of an inappropriate type.
- **NameError** is thrown when an object could not be found.
- **KeyboardInterrupt** is thrown when the user hits the interrupt key (normally Control-C) during the execution of the program.