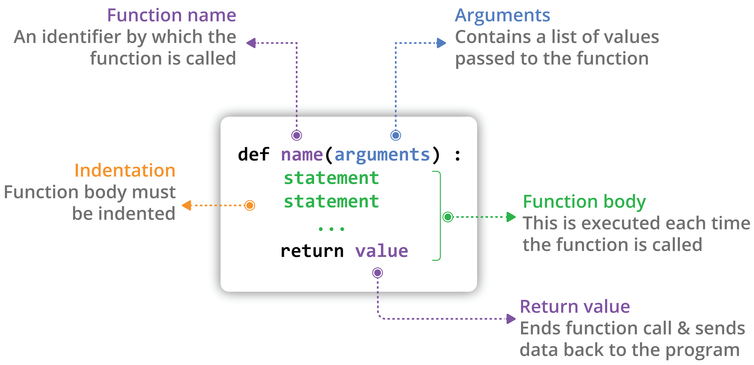
Day 8

Python for Data Science

Functions in Python

- In Python, a function is a group of related statements that performs a specific task.
- Functions help break our program into smaller and modular chunks.
- As our program grows larger and larger, functions make it more organized and manageable.
- Furthermore, it avoids repetition and makes the code reusable.
- There are two types of functions :
- Pre-defined functions
- User defined functions
- In Python a function is defined using the def keyword followed by the function name and parentheses ().
- Keyword def that marks the start of the function header.
- A function name to uniquely identify the function.
- Function naming follows the same rules of writing identifiers in Python.
- Parameters (arguments) through which we pass values to a function. They are optional.
- A colon (:) to mark the end of the function header.
- Optional documentation string (docstring) to describe what the function does.
- One or more valid python statements that make up the function body.
- Statements must have the same indentation level (usually 4 spaces).
- An optional return statement to return a value from the function.




```
# Take a function sample
# Mathematical operations in a function
def process(x):
  y1 = x-8
  y2 = x+8
  y3 = x*8
  y4 = x/8
  y5 = x%8
  y6 = x//8
  print(f'If you make the above operations with {x}, the results
will be {y1}, {y2}, {y3}, {y4}, {y5}, {y6}.')
  return y1, y2, y3, y4, y5, y6
process(5)
```

If you make the above operati ons with 5, the results will be -3, 13, 40, 0.625, 5, 0.
(-3, 13, 40, 0.625, 5, 0)

You can request help using help() function

```
help(process)
```

Help on function process in module __main__:
process(x)
  # Take a function sample
  # Mathematical operations in a function

Call the function again with the number 3.14

```
process(3.14)
```

If you make the above operati ons with 3.14, the results will be
-4.859999999999999, 11.14, 25.12, 0.3925, 3.14, 0.0.
(-4.859999999999999, 11.14, 25.12, 0.3925, 3.14, 0.0)

Functions with multiple parameters

```
# Define a function with multiple elements
def mult(x, y):
  z = 2*x + 5*y + 45
  return z
output = mult(3.14, 1.618) # You can yield the output by assigning
to a variable
print(output)
print(mult(3.14, 1.618)) # You can obtain the result directly
mult(3.14, 1.618) # This is also another version
```

```
59.370000000000005
59.370000000000005
59.370000000000005
```

```
# Call again the defined function with different arguments
print(mult(25, 34))
```

```
265
```

Variables

- The input to a function is called a formal parameter.
- A variable that is declared inside a function is called a local variable.
- The parameter only exists within the function (i .e. the point where the function starts and stops).
- A variable that is declared outside a function definition is a global variable, and its value is accessible and modifiable throughout the program.

```
# Define a function
def function(x):
  # Take a local variable
  y = 3.14
  z = 3*x + 1.618*y
  print(f'If you make the above operations with {x}, the results
will be {z}.')
  return z
with_golden_ratio = function(1.618)
print(with_golden_ratio)
```

```
If you make the above operations with 1.618, the results will be 9.934520000000001.
9.934520000000001
```

```
# It starts the gloabal variable
a = 3.14
# call function and return function
y = function(a)
print(y)
```

```
If you make the above operations with 3.14, the results will be 14.500520000000002.
14.500520000000002
```

```
# Enter a number directly as a parameter
function(2.718)
```

```
If you make the above operations with 2.718, the results will be 13.23452.
13.23452
```

Without return statement, the function returns None

```
# Define a function with and without return statement
def msg1():
  print('Hello, Python!')
def msg2():
  print('Hello, World!')
  return None
msg1()
msg2()
```

```
Hello, Python!
Hello, World!
```

```
# Printing the function after a call indicates a None is the de-
fault return statement.
# See the following prontings what functions returns are.
print(msg1())
print(msg2())
```

```
Hello, Python!
None
Hello, World!
None
```

Concatetantion of two strings

```
# Define a function
def strings(x, y):
  return x + y
# Testing the function 'strings(x, y)'
strings('Hello', ' ' 'Python')
```

'Hello Python'

Simplicity of functions

```
# The following codes are not used again.
x = 2.718
y = 0.577
equation = x*y + x+y - 37
if equation>0:
  equation = 6
else:
  equation = 37
equation
```

37

```
# The following codes are not used again.
x = 0
y = 0
equation = x*y + x+y - 37
if equation<0:
   equation = 0
else:
  equation = 37
equation
```

0

```
# The following codes can be write as a function.
def function(x, y):
  equation = x*y + x+y - 37
  if equation>0:
    equation = 6
  else:equation = 37
  return equation
x = 2.718
y = 0.577
function(x, y)
```

37

```
# The following codes can be write as a function.
def function(x, y):
  equation = x*y + x+y - 37
  if equation<0:
    equation = 6
  else:equation = 37
  return equation
x = 0
y = 0
function(x, y)
```

6

Predefined functions like print(), sum(), len(), min(), max(), input()

```
# print() is a built-in function
special_numbers = [0.577, 2.718, 3.14, 1.618, 1729, 6, 28, 37]
print(special_numbers)
```

[0.577, 2.718, 3.14, 1.618, 1729, 6, 28, 37]

```
# The function sum() add all elements in a list or a tuple
sum(special_numbers)
```

1808.053

```
# The function len() gives us the length of the list or tuple
len(special_numbers)
```

8

Using conditions and loops in functions

```
# Define a function including conditions if/else
def fermentation(microorganism, substrate, product, activity):
  print(microorganism, substrate, product, activity)
  if activity < 1000:
    return (f'The fermentation process was unsuccessful with the {product} activity of {activity} U/mL from {substrate}')
  else:
    return (f'The fermentation process was successful with the {product} activity of {activity} U/mL from {substrate}')
result1 = fermentation('Aspergillus niger', 'molasses', 'inulinase', 1800)
print(result1)
print()
result2 = fermentation('Aspergillus niger', 'molasses', 'inulinase', 785)
print(result2)
```

Aspergillus niger molasses inulinase 1800
The fermentation process was successful with the inulinase activity of 1800 U/mL from molasses

Aspergillus niger molasses inulinase 785
The fermentation process was unsuccessful with the inulinase activity of 785 U/mL from molasses

```
# Define a function using the loop 'for'
def fermentation(content):
  for parameters in content:
    print(parameters)
content = ['Stirred-tank bioreactor' ,'30°C temperature', '200 rpm agitation speed', '1 vvm aeration', '1% (v/v) inoculum', 'pH control at 5.0']
fermentation(content)`
```

Stirred-tank bioreactor
30°C temperature
200 rpm agitation speed
1 vvm aeration
1% (v/v) inoculum
pH control at 5.0

Adjusting default values of independent variables in functions

```
# Define a function adjusting the default value of the variable
def rating_value(rating = 5.5):
  if rating < 8:
    return f'You should not watch this film with the rating value of {rating}'
  else:
    return f'You should watch this film with the rating value of {rating}'
print(rating_value())
print(rating_value(8.6))
```

You should not watch this film with the rating value of 5.5
You should watch this film with the rating value of 8.6

Global Variables

- Variables that are created outside of a function (as in all of the examples above) are known as global variables.
- Global variables can be used by everyone, both inside of functions and outside.

```
# Define a function for a global variable
language = 'Python'
def lang(language):
  global_var = language
  print(f'{language} is a program language.')
lang(language)
lang(global_var)

"""
The output gives a NameError, since all variables in the function are local variables,
so variable assignment is not persistent outside the function.
```

```
Python is a program language.
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [23], in <cell line: 7>()
      5     print(f'{language} is a program language.')
      6 lang(language)
----> 7 lang(global_var)
      9 """
     10 The output gives a NameError, since all variables in the function
        are local variables,
     11 so variable assignment is not persistent outside the function.
     12 """

NameError: name 'global_var' is not defined
```

```
# Define a functi on for a global variable
language = 'JavaScript'
def lang(language):
  global global_var
  global_var = 'Python'
  print(f'{language} is a programing language.')
lang(language)
lang(global_var)
```

JavaScript is a programing language.
Python is a programing language.

Variables in functions

- The scope of a variable is the part of the program to which that variable is accessible.
- Variables declared outside of all function definitions can be accessed from anywhere in the program.
- Consequently, such variables are said to have global scope and are known as global variables.

```
process = 'Continuous fermentation'
def fermentation(process_name):
  if process_name == process:
    return '0.5 g/L/h.'
  else:
    return '0.25 g/L/h.'
print('The productiovity in continuous fermentation is', fermenta-
tion('Continuous fermentation'))
print('The productiovity in batch fermentation is', fermenta-
tion('Batch fermentation'))
print('Continuous fermentation has many advantages over batch fer-
mentation.')
print(f'My favourite process is {process}.')
```

The productiovity in continuous fermentation is 0.5 g/L/h.
The productiovity in batch fermentation is 0.25 g/L/h.
Continuous fermentation has many advantages over batch fermentation.
My favourite process is Continuous fermentation.

```
# If the variable 'process' is deleted, it returns a NameError as
follows
del process
# Since the variable 'process' is deleted, the following function
is an example of local variable
def fermentation(process_name):
  process = 'Continuous fermentation'
  if process_name == process:
    return '0.5 g/L/h.'
  else:
    return '0.25 g/L/h.'
print('The productiovity in continuous fermentation is', fermenta-
tion('Continuous fermentation'))
print('The productiovity in batch fermentation is', fermenta-
tion('Batch fermentation'))
print('Continuous fermentation has many advantages over batch fer-
mentation.')
print(f'My favourite process is {process}.')
```

The productiovity in continuous fermentation is 0.5 g/L/h.
The productiovity in batch fermentation is 0.25 g/L/h.
Continuous fermentation has many advantages over batch fermentation.

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [26], in <cell line: 13>()
     11 print('The productiovity in batch fermentation is', fermenta
        tion('Batch fermentation'))
     12 print('Continuous fermentation has many advantages over batch fer
        mentation.')
---> 13 print(f'My favourite process is {process}.')

NameError: name 'process' is not defined
```

```
# When the global variable and local variable have the same name:
process = 'Continuous fermentation'
def fermentation(process_name):
  process = 'Batch fermentation'
  if process_name == process:
    return '0.5 g/L/h.'
  else:
    return '0.25 g/L/h.'
print('The productiovity in continuous fermentation is', fermenta-
tion('Continuous fermentation'))
print('The productiovity in batch fermentation is', fermenta-
tion('Batch fermentation'))
print(f'My favourite process is {process}.')
```

The productiovity in continuous fermentation is 0.25 g/L/h.
The productiovity in batch fermentation is 0.5 g/L/h.
My favourite process is Continuous fermentation.

(args) and/or (*args) and Functions

When the number of arguments are unknown for a function, then the arguments can be packet into a tuple or a dictionary

```
# Define a function regarding a tuple example
def function(*args):
  print('Number of elements is', len(args))
  for element in args:
    print(element)
function('Aspergillus niger', 'inulinase', 'batch', '1800 U/mL acti
vity')
print()
function('Saccharomyces cerevisia', 'ethanol', 'conti nuous', '45%
yield', 'carob')
```

Number of elements is 4
Aspergillus niger
inulinase
batch
1800 U/mL acti vity

Number of elements is 5
Saccharomyces cerevisia
ethanol
conti nuous
45% yield
Carob

```
# Another example regarding 'args'
def total(*args):
  total = 0
  for i in args:
    total += i
  return total
print('The total of the numbers is', total(0.577, 2.718, 3.14,
1.618, 1729, 6, 37))
```

The total of the numbers is 1780.053

```
# Define a function regarding a dictionary example
def function(**args):
  for key in args:
    print(key, ':', args[key])
function(Micoorganism='Aspergillus niger', Substrate='Molasses',
Product='Inulinase', Fermentation_mode='Batch', Activity ='1800
U/ml')
```

Micoorganism : Aspergillus niger
Substrate : Molasses
Product : Inulinase
Fermentation_mode : Batch
Activity : 1800 U/ml

```
# Define a function regarding the addition of elements into a list
def addition(nlist):
  nlist.append(3.14)
  nlist.append(1.618)
  nlist.append(1729)
  nlist.append(6)
  nlist.append(37)
my_list= [0.577, 2.718]
addition(my_list)
print(my_list)
print(sum(my_list))
print(min(my_list))
print(max(my_list))
print(len(my_list))
```

[0.577, 2.718, 3.14, 1.618, 1729, 6, 37]
1780.053
0.577
1729
7

Doctsting in Functions

```
# Define a function
def addition(x, y):
  """The following function returns the sum of two parameters."""
  z = x+y
  return z
print(addition.__doc__)
print(addition(3.14, 2.718))
```

The following function returns the sum of two parameters.
5.8580000000000005

Recursive functions

```
# Calculating the factorial of a certain number.
def factorial(number):
  if number == 0:
    return 1
  else:
    return number*factorial(number-1)
print('The value is', factorial(6))
```

The value is 720

```
# Define a function that gives the total of the first ten numbers
def total_numbers(number, sum):
  if number == 11:
    return sum
  else:
    return total_numbers(number+1, sum+number)
print('The total of first ten numbers is', total_numbers(1, 0))
```

The total of first ten numbers is 55

Nested functions

```
# Define a function that add a number to another number
def added_num(num1):
  def incremented_num(num1):
    num1 = num1 + 1
    return num1
  num2 = incremented_num(num1)
  print(num1, '------->>', num2)
added_num(25)
```

25 -------->> 26