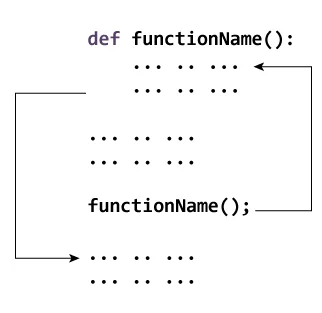
# Python Functions

Fungsi Python adalah sekelompok statement terkait yang melakukan tugas tertentu.

Fungsi membantu memecah program menjadi potongan yang lebih kecil dan modular. Seiring bertambahnya code yang kita tulis, membuatnya sebagai beberapa fungsi dapat lebih mudah dalam mengorganisir dan menggunakannya kembali.

Selain itu, pembuatan function dapat menghindari pengulangan dan membuat kode dapat digunakan kembali.

Bentuk dari suatu function:

| def **function\_name**(parameters):  """docstring"""  statement(s) |
| --- |

Jika kita terapkan untuk membuat program yang mencetak nama, maka kodenya seperti dibawah ini:

| def **greet**(name):  """  This function greets to  the person passed in as  a parameter  """  print("Hello, " + name + ". Good morning!") |
| --- |

| >>> greet('Angga') Hello, Angga. Good morning! |
| --- |

## Return Statement

Pernyataan return digunakan untuk membuat suatu fungsi mengembalikan nilai. Contohnya:

| def **my\_function**(x):  return 5 \* x  print(my\_function(3)) |
| --- |

Statement ini berisi ekspresi yang nilainya dikembalikan. Jika tidak ada ekspresi dalam statement atau pernyataan pengembalian nilai itu tidak ada di dalam suatu fungsi, maka fungsi tersebut akan mengembalikan objek **None**. Contohnya kita gunakan fungsi greet diatas:

| **>>>** print(greet("May")) Hello, May. Good morning! None |
| --- |

Contoh fungsi dengan return value:

| def **absolute\_value**(num):  """This function returns the absolute  value of the entered number"""   if num >= 0:  return num  else:  return -num   print(absolute\_value(2))  print(absolute\_value(-4)) |
| --- |

### Return multiple values

Fungsi di Python juga memungkinkan kita untuk mengembalikan banyak values dari eksekusi fungsi tersebut, misalnya:

| def **return\_multiple**():  return 1,2,3  a,b,c = return\_multiple() print(a) print(b) print(c) |
| --- |

Kita harus meng-unpacking value tersebut sesuai dengan jumlah return yang dihasilkan, jika tidak maka akan terjadi error.

Kita juga bisa menggunakan unpacking operator jika ingin mengembalikan return value sebagai suatu list.

| *# Returning Multiple Values of Different Length Variables* def **return\_multiple**():  return 1, 2, 3 a, \*b = return\_multiple() print(f'{a=}') print(f'{b=}') *# Returns:* *# a=1* *# b=[2, 3]* |
| --- |

### 

## Scope dan Lifetime dari variables

Scope variable adalah bagian dari program di mana variabel tersebut dapat dikenali. Parameter dan variabel yang didefinisikan di dalam suatu fungsi tidak dapat diakses dari luar fungsi. Oleh karena itu, mereka memiliki ruang lingkup lokal atau local scope.

Lifetime dari variabel adalah periode dimana variabel ada dalam memori. Lifetime variabel di dalam suatu fungsi adalah selama fungsi dijalankan. Suatu fungsi tidak mengingat nilai variabel dari pemanggilan sebelumnya.

| def **my\_func**():  x = 10  print("Value inside function:",x)  x = 20 my\_func() print("Value outside function:",x) |
| --- |

## Pass Statement

Definisi dari fungsi tidak boleh kosong, tetapi jika karena alasan tertentu kita memiliki definisi fungsi kosong, masukkan pernyataan **pass** untuk menghindari error.

| def **myfunction**():  pass |
| --- |

## Arguments

Fungsi python dapat menerima argumen, argumen ditetapkan di dalam kurung setelah nama fungsi tersebut. Kita dapat menambahkan argumen sebanyak yang kita inginkan, cukup pisahkan dengan koma.

| def **greet**(name, message):  """This function greets to  the person with the provided message"""  print("Hello", name + ', ' + msg)  greet("Angga", "Good morning!") |
| --- |

| Hello Angga, Good morning! |
| --- |

Disini fungsi greet memiliki dua parameter, name dan message. Karena kita memasukkan argumen sesuai dengan yang diminta fungsi tersebut, maka fungsi dapat dieksekusi dan berjalan dengan baik. Kita hanya 1 atau lebih dari 2 argumen yang dimasukkan, maka akan terjadi error ketika kita mengeksekusi fungsi tersebut.

### Python Default Arguments

Argumen dari suatu fungsi dapat memiliki nilai default. Kita dapat memberikan nilai default ke argumen dengan menggunakan operator =.

| def **greet**(name, message="Good morning!"):  """  This function greets to  the person with the  provided message.   If the message is not provided,  it defaults to "Good  morning!"  """   print("Hello", name + ', ' + msg)   greet("Angga") greet("Budi", "How do you do?") |
| --- |

| Hello Angga, Good morning! Hello Budi, How do you do? |
| --- |

Urutan pemanggilan dari argumen harus sesuai dengan urutan yang diminta oleh fungsi tersebut, jika tidak maka akan terjadi SyntaxError.

### Python Keyword Arguments

Saat kita memanggil fungsi dengan beberapa nilai, nilai-nilai ini ditugaskan ke argumen sesuai dengan posisinya.

Sebagai contoh, pada fungsi di atas greet(), ketika kita menyebutnya sebagai greet("Budi", "How are you?"), nilai "Budi" diberikan ke argumen **name** dan juga "How are you? " ke argumen **message**.

Python memungkinkan fungsi dipanggil menggunakan **keyword** **argument**. Ketika kita memanggil fungsi dengan cara ini, urutan (posisi) argumen dapat diubah atau tidak ada urutan pasti. Pemanggilan ke fungsi berikut semuanya valid dan menghasilkan hasil yang sama.

| *# 2 keyword arguments* greet(name = "Budi",message = "How do you do?")  *# 2 keyword arguments (out of order)* greet(message = "How do you do?",name = "Budi")   1 positional, 1 keyword argument greet("Budi", message = "How do you do?") |
| --- |

### Python Arbitrary Arguments, \*args

Jika Anda tidak tahu berapa banyak argumen yang akan dimasukkan ke fungsi, tambahkan \* sebelum nama parameter dalam definisi fungsi.

Dengan cara ini fungsi akan menerima tuple argumen, dan dapat mengakses item yang sesuai

| def **greet**(\*names):  """This function greets all  the person in the names tuple."""   *# names is a tuple with arguments*  for name in names:  print("Hello", name)   greet("Angga", "Steve", "Joni") |
| --- |

| Hello Monica Hello Luke Hello Steve Hello John |
| --- |

### Python Arbitrary Keyword Arguments, \*kwargs

Jika Anda tidak tahu berapa banyak keyword argumen yang akan diteruskan ke fungsi, tambahkan dua tanda bintang: \*\* sebelum nama parameter dalam definisi fungsi.

Dengan cara ini fungsi akan menerima dict sebagai argumen, dan dapat mengakses item yang sesuai:

| def **greet**(\*\*kwargs):  """This function greets a person with the full name."""  print("Hello", kwargs["first\_name"] + ', ' + kwargs["last\_name"])  greet(first\_name="Angga", last\_name="Pradikta") |
| --- |

## Python Recursion

Rekursi adalah proses mendefinisikan sesuatu dalam hal itu sendiri.

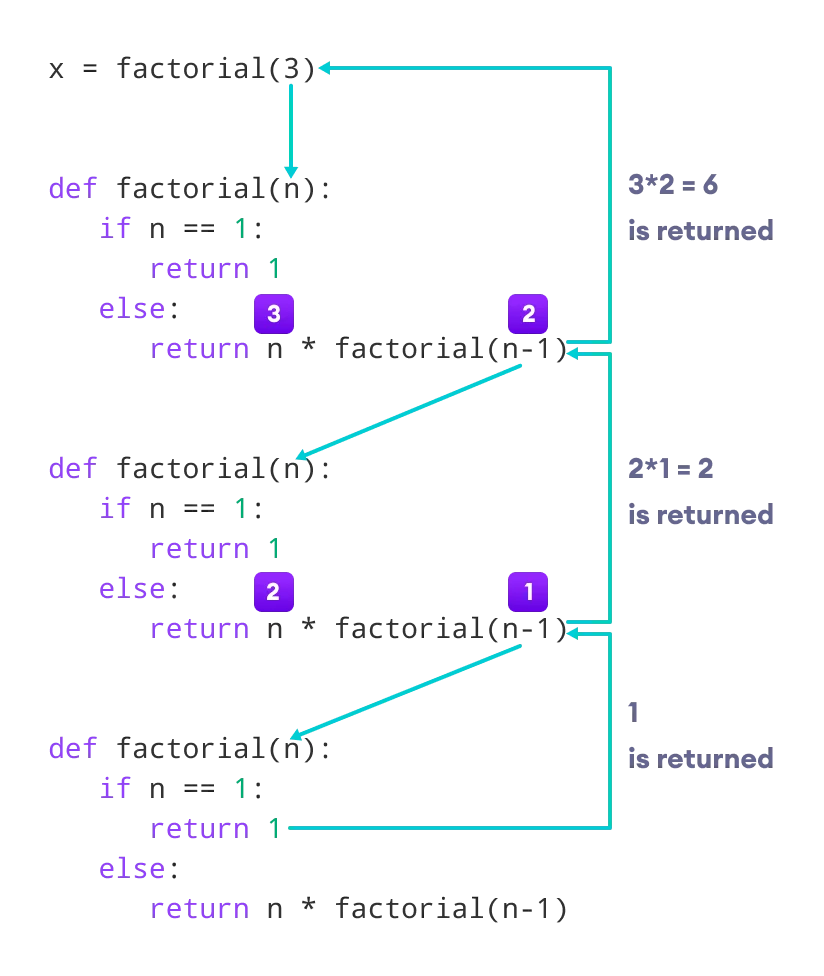
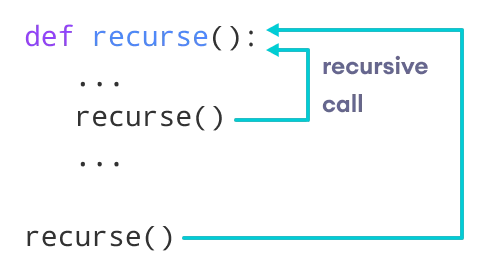
Contohnya seperti menempatkan dua cermin yang saling berhadapan. Objek apa pun di antara mereka akan dipantulkan secara rekursif.

### Python Recursive Function

Suatu fungsi dapat memanggil fungsi lainnya. Bahkan dimungkinkan bagi fungsi untuk memanggil dirinya sendiri. Jenis konstruksi ini disebut sebagai fungsi rekursif.

Berikut ini adalah contoh fungsi rekursif untuk mencari faktorial suatu bilangan bulat.

Faktorial suatu bilangan adalah hasil kali semua bilangan bulat dari 1 sampai dengan bilangan tersebut. Misalnya, faktorial dari 6 (dilambangkan dengan 6!) adalah 1\*2\*3\*4\*5\*6 = 720.



| def **factorial**(x):  """This is a recursive function  to find the factorial of an integer"""   if x == 1:  return 1  else:  return (x \* factorial(x-1))  num = 3 print("The factorial of", num, "is", factorial(num)) |
| --- |

| The factorial of 3 **is** 6 |
| --- |

Rekursi diatas berakhir ketika jumlahnya berkurang sampai menjadi 1. 1 disini disebut sebagai kondisi dasar. Setiap fungsi rekursif harus memiliki kondisi dasar yang menghentikan rekursi atau fungsi akan memanggil dirinya sendiri tanpa batas.

Python membatasi jumlah rekursi untuk membantu menghindari rekursi tak terbatas, secara default, jumlah rekursi maksimum adalah 1000.

Jika batas terlampaui, itu menghasilkan RecursionError.

## Anonymous atau Lambda Function

Fungsi anonim adalah fungsi yang didefinisikan tanpa nama.

Sementara fungsi normal didefinisikan menggunakan kata kunci def, fungsi anonim didefinisikan menggunakan kata kunci lambda.

Oleh karena itu, fungsi anonim juga disebut fungsi lambda.

| *# Example of lambda functions* double = lambda x: x \* 2  print(double(5)) |
| --- |

Di sini x adalah argumen dan x \* 2 adalah ekspresi yang dievaluasi dan dikembalikan. Fungsi ini tidak memiliki nama. Ini mengembalikan objek fungsi yang ditugaskan ke pengidentifikasi double. Code diatas sama seperti:

| def **double**(x):  return x \* 2 |
| --- |

### Penggunaan Lambda Function

Kita dapat menggunakan fungsi lambda ketika membutuhkan fungsi tanpa nama untuk waktu yang singkat.

Keuntungan lambda dapat ditampilkan saat kita menggunakannya fungsi anonim di dalam fungsi lain. Katakanlah kita memiliki definisi fungsi yang mengambil satu argumen, dan argumen itu akan dikalikan dengan angka yang tidak diketahui, misalnya:

| def **myfunc**(n):  return lambda a : a \* n  mydoubler = myfunc(2)  print(mydoubler(11)) |
| --- |

Fungsi Lambda juga dapat digunakan bersama dengan fungsi bawaan seperti filter(), map() dll. Ini sangat berguna saat nanti kita berinteraksi dengan Python Spark.

| *# Program to filter out only the even items from a list* my\_list = [1, 5, 4, 6, 8, 11, 3, 12]  new\_list = list(filter(lambda x: (x%2 == 0) , my\_list))  print(new\_list) |
| --- |

Fungsi dipanggil dengan semua item dalam list dan list baru dihasilkan yang berisi item yang memenuhi kondisi True. Sama seperti condition

| *# Program to double each item in a list using map()*  my\_list = [1, 5, 4, 6, 8, 11, 3, 12]  new\_list = list(map(lambda x: x \* 2 , my\_list))  print(new\_list) |
| --- |

Map menghasilkan list baru dari list yang sudah dimodifikasi

## Local Variables

Variabel yang dideklarasikan di dalam suatu fungsi dalam lingkup lokal dikenal sebagai variabel lokal.

| def **foo**():  y = "local"  foo() print(y) |
| --- |

| NameError: name 'y' **is** **not** defined |
| --- |

Output menunjukkan error karena kita mencoba mengakses variabel lokal y dalam lingkup global sedangkan variabel lokal hanya berfungsi di dalam foo() atau lingkup lokal.

Biasanya, kita dapat mendeklarasikan variabel di dalam fungsi untuk membuat variabel lokal.

| def **foo**():  y = "local"  print(y)  foo() |
| --- |

| local |
| --- |

## Global and local variables

Menggunakan global dan local variable bersamaan

| x = "global"  def **foo**():  global x  y = "local"  x = x \* 2  print(x)  print(y)  foo() |
| --- |

| **global** **global**  local |
| --- |

Menggunakan global dan local variable dengan nama yang sama

| ​​x = 5  def **foo**():  x = 10  print("local x:", x)   foo() print("global x:", x) |
| --- |

| local x: 10 **global** x: 5 |
| --- |

## Nonlocal Variables

Variabel nonlokal digunakan dalam nested function yang cakupan lokalnya tidak ditentukan. Yang berarti bahwa variabel tidak boleh berada dalam lingkup lokal maupun global.

| def **outer**():  x = "local"   def **inner**():  nonlocal x  x = "nonlocal"  print("inner:", x)   inner()  print("outer:", x)  outer() |
| --- |

| inner: **nonlocal** outer: **nonlocal** |
| --- |

## Python Global Keyword

Keyword global memungkinkan kita untuk mengubah variable di luar cakupan saat ini. Keyword global digunakan untuk membuat variable global dan membuat perubahan pada variabel dalam scope lokal.

| c = 1 *# global variable*  def **add**():  print(c)  add() |
| --- |

Aturan dasar untuk Keyword global di Python adalah:

* Saat kita membuat variabel di dalam suatu fungsi, itu akan memiliki scope lokal secara default.
* Saat kita mendefinisikan variabel di luar fungsi, itu akan memiliki global secara default. Kita tidak harus menggunakan keyword global.
* Kita menggunakan keyword global untuk membaca dan menulis variabel global di dalam suatu fungsi.
* Penggunaan keyword global di luar fungsi tidak berpengaruh.

| c = 1 *# global variable*   def **add**():  c = c + 2 *# increment c by 2*  print(c)  add() |
| --- |

| UnboundLocalError: local variable 'c' referenced before assignment |
| --- |

Ini menghasilkan error karena kita mencoba memodifikasi variable global c, supaya eksekusi ini berhasil maka **global c** harus ditambahkan di dalam fungsi add()

## if \_\_name\_\_ == "\_\_main\_\_":

Ada beberapa keuntungan ketika kita menambahkan line diatas saat mengeksekusi kode di Python. Terutama saat kita memiliki beberapa fungsi yang harus berinteraksi, misalnya:

new\_module.py

| def **new\_function**(a):  print(f"new\_function got {a}")  print("This is my\_module!") new\_function(33) |
| --- |

start.py

| import new\_module  print("This is start.py!") new\_module.new\_function(2) |
| --- |

| This **is** new\_module! new\_function got 33 This **is** start.py! new\_function got 2 |
| --- |

Padahal kita hanya mengeksekusi start.py dan hanya ingin menghasilkan output dari start.py, namun output dari fungsi yang kita re-use juga muncul, untuk menghindari hal ini maka kita dapat menulisnya seperti ini:

new\_module.py

| def **new\_function**(a):  print(f"new\_function got {a}")  if \_\_name\_\_ == "\_\_main\_\_":  print("This is new\_package!")  new\_function(33) |
| --- |

start.py

| import new\_module  if \_\_name\_\_ == "\_\_main\_\_":  print("This is start.py")  new\_module.new\_function(2) |
| --- |

| This **is** start.py new\_function got 2 |
| --- |

Variabel khusus \_\_name\_\_ di-assign sebagai nama file dari file yang sedang kita eksekusi. Fitur ini dapat digunakan untuk menjaga kode dari eksekusi saat mengimpor modul.