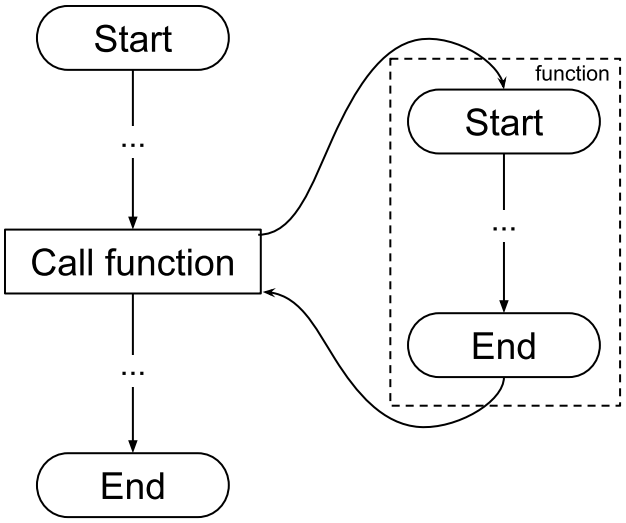
> Python workshop
>
> ***3-1. Functions Revisited***

# Introduction

## Recall: Function

**Function** allow us to define sequence of code to be re-used.




## Concepts: Function signature

When we define a function, we are specifying:

- The **input** to the function;
- The **body** of the function, which produce the **output** of a function

The **input** and **output** of a function is the **signature** of the function.

- A well defined function must be self-contained, i.e., it works in its own scope and is not affected by the program state.
- Python do not require a function signature to be defined, but we can write comments to indicate it.
- Formally we can use **docstrings** to document a function.

```
def add(a, b):
    """Return the result of adding parameter a and b."""
```

```
    return a + b
```

Reference: https://www.python.org/dev/peps/pep-0257/

## Recursion

A function can call the function itself, which makes it a recursive function. In that case the function must have a terminal condition.

In [1]:
```
def listSum(myList):
    # terminal condition
    if len(myList) == 0:
        return 0

    return myList[0] + listSum(myList[1:])

print(listSum([1,2,3]))
```

```
6
```

# Quick Quiz

What is the output of the following program?

```
def a(p):
  return p * 3

def b(q):
  return q + 1

x = a(b(a(b(1))))
print(x)
```

| A | B | C | D |
|---|---|---|---|
| 7 | 12 | 13 | 21 |

# Demonstration 3-1

Implement function `comb(n, r)` that calculates **combination** $\binom{n}{r}$, or $nCr$, using the recursive formular:

$$\binom{n}{r} = \begin{cases} 1 & \text{for } r = 0 \text{ or } n = r \\ \binom{n-1}{r} + \binom{n-1}{r-1} & \text{otherwise} \end{cases}$$

Your program should read the two values, `n` and `r`, as integers and outputs the result.

Sample input/output

| Input | Output |
| --- | --- |
| 5<br>2 | 10 |
| 7<br>3 | 35 |
| 13<br>7 | 1716 |

# Self-learning topics

Functions

- Function return
- Variable scoping
- Function parameters

## Function return

A function can **return** a value for future use. This serves as the output of a function.

```
def add(x, y):
    return x + y
```

This defines a function `add()` that accepts two parameters, `x` and `y`, and returns the result of `x + y`. The result could be captured with a variable, or be used immediately.

In [2]:
```
def add(x, y):
    return x + y

z = add(1, 2)
print("1 + 2 is", z)
print("1 + 2 is", add(1, 2))
```

```
1 + 2 is 3
1 + 2 is 3
```

### None and pass

Remember that an empty block can be defined using `pass`. If a function did not return anything, the value of `None` will then be returned.

In [3]:
```
def func():
    pass

print(func())
```

```
None
```

### Return value packing and unpacking

Remember value packing and unpacking when we discuss tuple? Function return could do the same.

In [4]:
```
def swap(a, b):
    return b, a

a = 10
b = 20
a, b = swap(a, b)
print(a, b)
```

```
20 10
```

Here, `b, a` is **packed** into a tuple, and returned. The tuple will then be **unpacked** automatically if we specify a list of variables to receive the returned value(s).

## Variable scope

Scope of a variable affects the availability of the variable.

Variable defined outside of a function has a **global** scope, which be accessed anywhere in the program.

In [5]:
```
x = 1
def func():
    print(x)

func()
```

```
1
```

If we assigning a value to a variable in a function, a **local** variable is created. A **local** variable cannot be used outside a function.

In [6]:
```
def func():
    xxx = 1
    print(xxx)

func()
print(xxx) # Error!
```

```
1
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[6], line 6
      3     print(xxx)
      5 func()
----> 6 print(xxx) # Error!

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

A variable can either be global or local in a function but not both. The following code will cause an error:

In [7]:
```
x = 1
def func():
    print(x)
    x = 2
    print(x)

func() # Error!
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Cell In[7], line 7
      4     x = 2
      5     print(x)
----> 7 func() # Error!

Cell In[7], line 3, in func()
      2 def func():
----> 3     print(x)
      4     x = 2
      5     print(x)

UnboundLocalError: cannot access local variable 'x' where it is not associated with
a value
```

As `x` is being assigned in the function, `x` must be a local variable. The first print will fail because local variable `x` is not assigned yet.

In terms of good program design, functions are not supposed to update global variables (Remember that function should be self-contained if possible). If we really need to update a global variable in a function, we must declare the variable **global** in the function using the `global` keyword:

In [9]:
```
x = 1
def func():
    global x
    print(x)
    x = 2
    print(x)

func()
```

```
1
2
```

Try to avoid using global variables at all cost. There are a few exceptions (e.g., constants, etc.), but in most cases there are better choices.

# Function parameters

## Parameters and arguments

Recall one of our previous examples:

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

hello('David')
```

- Our `hello()` function is defined with 1 parameter, we need to specify one value as argument when we call the function.
- The term **parameter** refers to the variable name(s) defined in the function, the term **argument** refers to the value passed into a function when we use it.

## Function arguments

We can define any number of parameters for a function.

```
def hello0():
    print('Hello world')

def hello1(name):
    print('Hello', name)

def hello2(name, message):
    print('Hello', name)
    print(message)
```

## Specifying parameters

When there are multiple parameters, the values are specified in order:

In [10]:
```
def hello2(name, message):
    print('Hello', name)
    print(message)

hello2('David', 'How are you?')
```

```
Hello David
How are you?
```

### Passing a list

When a mutable object (e.g., list) is passed into a function, the effect is the same as if assignment operator `=` is used. The code:

In [11]:
```
def f(myList2):
  myList2[0] = 4

myList = [1, 2, 3]
f(myList)
print(myList)
```

```
[4, 2, 3]
```

will have the same result as:

In [12]:
```
myList = [1, 2, 3]
myList2 = myList
myList2[0] = 4
print(myList)
```

```
[4, 2, 3]
```

# Exploration task

What happens if you collect the return value from a function, but the function did not return anything?

# Self-evaluation exercises (3-1)

## Quiz 3-1

Consider the program below.

```
def alpha(x, y=1):
  print(x + y, end="")
```

1. What is the output if `print(alpha(100))` is executed right after the above program?
2. If `print(alpha(100))` is the intended usage of the function `alpha`, what should the body of the function be? Remove all **unnecessary** spaces in your answer.

## Exercise 3-1

Given the following code as the main program, complete two functions `readList()` and `findNo()` as explained in the next page.

```
myList = readList()
number = int(input())
print(findNo(myList, number))
```

- `readList()` : Read numbers from user until zero is received. Return the list of numbers.
- `findNo(myList, number)` : return `True` if the number is in the list; else `False` .

## Challenge 3-1

Note: This exercise requires the use of **default arguments**, which is explained in the optional section **Advanced function parameters**, available after this exercise.

Implement your own `range()` function, name it `myRange()` which generate a list based on the input arguments.

- It must support one, two, or three arguments. Assuming that all arguments are non-negative.
- If you want to challenge yourselves, try to support negative values also.

You can compare your function with the output of `range()` function, to print the list of values generated by `range()` , convert it to a list first. For example:

```
print(myRange(10))
print(list(range(10)))

print(myRange(2, 10))
print(list(range(2, 10)))

print(myRange(2, 10, 3))
print(list(range(2, 10, 3)))
```

# Optional topics

- Advanced function parameters
- Argument packing and unpacking
- First-class functions
- Lambda functions

## Advanced function parameters

### Default arguments

We can set a **default** for some of the parameters. In this way, the default values will be used if the values are not specified when the function is called.

In [13]:
```
def hello(hello='hello', name='David', message='How are you'):
    print(hello, name+".", message)

hello()
```

```
hello David. How are you
```

In [14]:
```
hello('Hi')
```

```
Hi David. How are you
```

In [15]:
```
hello('Hi', 'Jason')
```

```
Hi Jason. How are you
```

In [16]:
```
hello('Hi', 'Jason', 'Welcome')
```

```
Hi Jason. Welcome
```

## Default argument limitations

Note that parameters with default values must be at the **end** of the argument list.

```
def hello(hello, name='David', message='How are you'):
    print(hello, name)
    print(message)
```

So this is incorrect:

```
def hello(hello, name='David', message):
    print(hello, name)
    print(message)
```

## Keyword arguments

We can choose to specify a value by **keyword**. All three function calls below produce the same result.

In [17]:
```
def hello(name='David', message='How are you'):
    print('Hello', name)
    print(message)

hello('David', 'How are you?')
hello('David', message = 'How are you?')
hello(name = 'David', message = 'How are you?')
```

```
Hello David
How are you?
Hello David
How are you?
Hello David
How are you?
```

## Keyword arguments limitations

Once a keyword argument is specified, all remaining values must be specified as keyword arguments.

So this is invalid:

In [18]:
```
def hello(name='David', message='How are you'):
    print('Hello', name)
    print(message)

hello(name = 'David', 'How are you?')
```

```
  Cell In[18], line 5
    hello(name = 'David', 'How are you?')
                                         ^
SyntaxError: positional argument follows keyword argument
```

## Keyword and Defaults

It is also possible to specify some arguments by keywords and leave the other using defaults.

In [19]:
```
def hello(hello='hello', name='David', message='How are you'):
    print(hello, name)
    print(message)

hello(name = 'Jason')
hello('Hi', message = 'Welcome')
```

```
hello Jason
How are you
Hi David
Welcome
```

# Argument packing and unpacking

## Argument unpacking

We can unpack a list of values into arguments using `*` operator.

In [20]:
```
def hello(hello='hello', name='David', message='How are you'):
    print(hello, name)
    print(message)
```

```
hello(*['Hi', 'Jason', 'Welcome!'])
```

```
Hi Jason
Welcome!
```

## Argument unpacking usage

It will be useful when we want to print a list:

In [21]:
```
myList = ['apple', 'banana', 'orange']
print('I like', end=" ")
print(*myList, sep=", ")
```

```
I like apple, banana, orange
```

## Keyword argument unpacking

We can also use a dictionary for keyword arguments, in this case we use the `**` operator instead.

In [22]:
```
def hello(hello='hello', name='David', message='How are you'):
    print(hello, name)
    print(message)

hello(**{'name': 'Jason', 'message': 'Welcome'})
```

```
hello Jason
Welcome
```

## Variable keyword arguments

We can define a parameter in the form of `**name` at the end of parameter list to consume any keyword arguments that is not handled in the list:

In [23]:
```
def listPrices(name='My Store', **prices):
    print('Listing prices for', name)
    for item in prices:
        print(item, ':', prices[item])

listPrices(**{'apple': 10, 'banana': 15, 'orange': 20})
```

```
Listing prices for My Store
apple : 10
banana : 15
orange : 20
```

In the case above, `prices` will be a dictionary of the keyword arguments.

## Variable arguments

We can define a parameter in the form of `*name` to consume any number of non-keyword arguments:

In [24]:
```
def func(a, b, *c):
    print(a, b, c)

func(1, 2)
func(1, 2, 3)
func(1, 2, 3, 4)
```

```
1 2 ()
1 2 (3,)
1 2 (3, 4)
```

### Variable arguments limitations

There can only be one `*name` in the parameter list. All parameter after that must be specified by keyword.

For example if the function is defined like the code below, then `c` must be specified by keyword:

In [25]:
```
def func(a, *b, c):
    print(a, b, c)

func(1, 2, c=3)
```

```
1 (2,) 3
```

`*name` and `**name` can be used together. In that case, `**name` must be placed at the end.

In [26]:
```
def func(*b, **c):
    print(b, c)

func(1, 2)
func(1, 2, x=1, y=2)
```

```
(1, 2) {}
(1, 2) {'x': 1, 'y': 2}
```

# First-class functions

Python functions are **first-class functions**, all function is treated as a value, assigned the a variable named by the function name. This is a feature very commonly seen in modern programming languages.

We can therefore assign a function to a variable:

In [27]:
```
def myFunc():
    print('This is myFunc')
```

```
myFunc2 = myFunc
myFunc2()
```

```
This is myFunc
```

## Namespace

Function name and variable name uses the same name space. If we define a variable of the same name as a function, we cannot use the function anymore.

For example, this will cause an error when executed:

In [28]:
```
def func():
    pass

func = 0
func()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[28], line 5
      2     pass
      4 func = 0
----> 5 func()

TypeError: 'int' object is not callable
```

## Function as arguments

Since function can be used as a variable, we can pass a function as an argument. The above below will calculate and print the sum of $1^2$ to $9^2$, which equals $285$.

In [29]:
```
def square(val):
    return val**2

def sumof(values, func):
    sum = 0
    for val in values:
        sum += func(val)
    return sum

print(sumof(range(1, 10), square))
```

```
285
```

## Local function

Similar to scope of variables, function can also be defined locally.

In [30]:
```
def func():
    def innerFunc(a):
        return a**2
```

```
    return innerFunc(10)

print(func())
```

```
100
```

### Function as returned value

We can also return a function using its name or return the variable name holding the function.

In [31]:
```
def func(choice):
    def innerFunc1(a):
        return a**2
    def innerFunc2(a):
        return a**3
    if choice == 'square':
        return innerFunc1
    elif choice == 'cube':
        return innerFunc2

print(func('square')(1000))
print(func('cube')(10))
```

```
1000000
1000
```

## Lambda functions

We can use the `lambda` keyword to define a simple anonymous function.

For example, a function that calculate the square of a variable is: `lambda x : x **2`.

- Following `lambda` is the argument list;
- After the colon `:` is the expression that gives the return value.
- Lambda function is limited to one single statement only due to its syntax.

One of the previous example can be modified to the code below.

In [32]:
```
def sumof(values, func):
    sum = 0
    for val in values:
        sum += func(val)
    return sum

print(sumof(range(1, 10), lambda val : val ** 2))
```

```
285
```

### Example: custom list sorting

Function argument is useful for function that allow customizable behaviours. For example, the `sort()` function of lists support one function argument to specify how values are interpreted when sorting the list. The code below sort a list in reverse order.

In [33]:
```
myList = [1, 4, 2, 5, 7, 6]
myList.sort()
print(myList)
myList.sort(key = lambda x : -x)
print(myList)
```

```
[1, 2, 4, 5, 6, 7]
[7, 6, 5, 4, 2, 1]
```