User-defined Functions

Function is a group of related statements that perform a specific task.

Why use function?

- Functions help break program into smaller and modular chunks.
- It supports code reusability and reduces repeative code.
- It make large program more organized and manageable.

1. Basic Syntax

```
def function_name(parameters):
    """docstring"""
    statement(s)
```

- Keyword `def` marks the start of a function.
- Every function has a name as uniquely identification.
- It may take in optional values, known as **parameters**, before running the code block.
- It may return vale as a result using `return` statement.
  - If no `return` statement, the function will `return None` at the end of the function.
- Optional documentation string (docstring) describes what the function does. It may be multiple lines of string.
- All statements must be equally identied, which is usually 4 spaces.

Note: Do not indent your program with mix of spaces and tabs.

What functions have used so far?

We have used a few functions when we learnt about Pyhton basic data types. Can you name any?

- How do you check data type of an object?
- How do you convert a value from one data type to another?
- How to you print out value(s) in console?

In [1]:
```
a = '123'
print(type(a))
b = int(a)
print(a, b)
```

```
<class 'str'>
123 123
```

2. Basic Functions

Let's get familiar with how to define a function

- Function with no input parameter nor return statement.
- Function with input parameter.
- Function with both input parameter and return statement.

Simplest Function that does nothing

Let's define a simplest function which does nothing.

- Keyword `pass` is used to indicate nothing to be done in the function body.

In [2]:
```
def do_nothing():
    pass
```

In [3]:
```
type(do_nothing)
```

Out[3]:
```
function
```

Hello World

Let's define a basic `hello_world` function without input parameter and return statement.

- It simply prints "Hell World".

In [4]:
```
def hello_world():
    print("Hello World")

print("Have a good day")
hello_world()
```

```
Have a good day
Hello World
```

Hello Who

Let's use **input parameter** to make the hello function more flexible.

- It take in an input value `who` , and prints "Hello {who}".

In [5]:
```
def hello_who(who):
    print("Hello {}".format(who))

hello_who("Singapore")
```

```
Hello Singapore
```

In [6]:
```
def add_them(a, b):
    return a+b

print(add_them(1,2))
print(add_them('hello ', 'world'))
print(add_them([1,2], ['a', 'b']))
```

```
3
hello world
[1, 2, 'a', 'b']
```

In [7]:
```
def mul_them(a, b):
    return a * b

print(mul_them(3, 4))
print(mul_them('hello', 4))
```

```
12
hellohellohellohello
```

How to call a function?

Function can be called/used in another function. Before a function can be used, it needs to be defined first. It can also be defined in another module and imported beofre use.

***Question:** What is wrong with following code?

In [8]:
```
def hello_twice():
    hello_who("World")
    hello_who("Singapore")

print(type(hello_twice))
hello_twice()
```

```
<class 'function'>
Hello World
Hello Singapore
```

Function with both Inputs and Return Statement

Let's define a function `my_max` which return the greater value of 2 input values.

In [9]:
```
def my_max(x, y):
    if x > y:
        return x
    else:
        return y

print(my_max(10, 20))

def my_max2(x, y):
    return x if x > y else y

my_max2(10, 20)
```

```
20
```

Out[9]:
```
20
```

In [10]:
```
x = 10
result = 'even' if x % 2 == 0 else 'odd'
print(result)
```

```
even
```

3. Using Functions and Modules (using IDLE)

It is common to put one or more functions in a file, which is called a Python **module**.

- All python files have extension of **.py**.

We will use **IDLE** in this session to get familiar with **modules**.

IDLE Working Directory

When IDLE starts, it presents a REPL (Read-Evaluate-Print-Loop), which is similar to how you run Python on command line. All command line have a current working directory.

Question: What is current working directory of IDLE?

To get and set current working directory of IDLE, we need the help of `os` module.

- `os.getcwd()` returns current working directory
- `os.chdir()` change current working directory

Start IDLE and try following:

- Verify that IDLE's current working directory is "C:\WINDOWS\system32"
- Change working directory to "C:\"

In [11]:
```
# >>> import os
# >>> os.getcwd()
# >>> os.chdir('C:\\')
# >>> os.getcwd()
```

Create New Module

Create a new Python file from IDEL using menu `File > New File`.

- Add following two functions to the file

```
def hello_once(name):
    print('Hello {}'.format(name))

def hello_many(items):
    for item in items:
        hello_once(item)
```

- Save file as `myhello.py`
- Press `F5` to run the module

Question: What is the current working directory of IDLE now?

- IDLE automatically change working directory to the folder containing the script which is run.

Import Module in IDLE

To import `myhello.py` module, use `import` statement without `.py` extension.

- After importing, we can use the functions from the module
- To call the function in the module, prefix the function by the module name. (Although it is ok to run without module prefix in IDLE)

In [12]:
```
# >>> import myhello
# >>> myhello.hello_once("World")
# >>> myhello.hello_many(["World", "Python"])
```

Re-import Module after Modification

- Modify the `hello_once` function in `myhello.py` file with following content. (Remember to save file)

```
def hello_once(name):
    print('Hello again {}'.format(name))
```

- Import the module again
- Run hello_once() function again

Question: Does the output reflect the code changes?

In [13]:
```
# >>> import myhello
# >>> myhello.hello_once('World')
```

Reload Imported Module

For each Python REPL session, module importing is only done once. To re-import a module, we need the help of `importlib` module.

- Upon import, the code in module is executed

In [14]:
```
# >>> import importlib
# >>> importlib.reload(myhello)
# >>> myhello.hello_once('World')
```

4. Module Import vs. Module Execution

We will try this session on Command Prompt or PowerShell.

Module Execution

Pyton file can be executed directly using Python executable on command line.

- Start a Windows Command Prompt or PowerShell. Go to folder containing `myhello.py`
- Run following command to execute the module

  ```
  $ python myhello.py
  ```

- Append following line to the end of `myhello.py` file. (Without any leading space on the line)

  ```
  hello_many(["World", "Python"])
  ```

- Execute the module again. Following are the printouts.

  ```
  Hello again World
  Hello again Python
  ```

Module Import

Let's try to import `myhello` module again.

- On Command Prompt or PowerShell, start a Python REPL session by executing `python` .
- Import `myhello` module.
  - Since this is a new Python REPL session, we dont need the help of `importlib` .
- **Question:** Is the module executed during import too?

Modules will be executed once during import. We would not like above printout when a module is imported in another module.

Question: How can we make a module suitable for both **execution** and **import**?

The __name__ Variable

Special attributes/variables in Python are enclosed by **double underscores**, e.g. `__name__` .

The `__name__` variable will have different value depending on how an enclosing module is used.

- It will be evaluated to `"__main__"` if the module is executed.
- It will be evaluated to module name if the module is imported.
- Append following line in `myhello.py` file.

```
print(__name__)
```

- Press CTRL+z (on Windows) to exit from Python REPL.
- Run following command to execute the module. What is the value of `__name__` ?

```
$ python myhello.py
```

- Start Python REPL by typing `python` .
- Import `myhello` module. What is the value of `__name__` ?

Use __name__ Variable in Module

We can make use of **name** variable to support both **module import** and **module execution**.

In your `myhello.py` file, following are the 2 last lines of code.

```
hello_many(["World", "Python"])
print(__name__)
```

Modify it to following code.

```
if __name__ == '__main__':
        hello_many(["World", "Python"])
```

Try to execute and import `myhello` module again.

- Press CTRL+z to exit Python REPL
- Run `python myhello.py`
- Start Python REPL by running `pyhton`
- Import `myhello` module

Question: how does `__name__` make a difference?

- It's a common practice to include a `if __name__ == '__main__':` session in python script so that it can be used for both module execution and module import.

More on Import Modules

Ther are several options to import a module.

- Import module and access functions through module

```
import myhello
```

- Import specific functions from module

```
from myhello import hello_once, hello_many
```

- Imoprt everything from a module

```
from myhello import *
```

Last option NOT recommended because you do not have a control on what is imported. It may potentially cause **namespace collashes**, if the imported module contains functions of same name as your own functions.

Command Line Arguments

When executing a python file, we may want to pass it some arguments, which is called **Command Line Arguments**.

To get the list of arguments passed to the script, we use `sys.argv` in `sys` module.

- `sys.argv` is the list of command-line arguments.
- `len(sys.argv)` is the number of command-line arguments.
- `sys.argv[0]` is the script name.

Question: How to get list of all arguments excluding script name?

Let's use them in our `myhello` module.

- Modify the `__main__` code block in `myhello.py` file as following

```
import sys
if __name__ == '__main__':
      print("Script name: {}".format(sys.argv[0]))
      if len(sys.argv) == 2:
              hello_once(sys.argv[1])
      if len(sys.argv) > 2:
              hello_many(sys.argv[1:])
```

- On Command Prompt or PowerShell, run following commands

```
$ python myhello.py
$ python myhello.py World
$ python myhello.py World Python
```

(Optional) What about Named Command Line Arguments?

If multple arguments to be passed to a script, it is less confusing if user can specify the name of each argument. Example, `python myhello.py --whom World` .

It's common to use named command arguments while executing scripts. To implement it, check out `argparse` module.

Reference: https://stackoverflow.com/questions/40001892/reading-named-command-arguments (https://stackoverflow.com/questions/40001892/reading-named-command-arguments)

5. Docstring

The first string after the function header is called **documentation string**, which is commonly called **docstring**.

Docstring serves as documentation for your function so that anyone who reads the function's docstring understands what the function does, without having to trace through all the code in the function definition.

- It is used to explain briefly what a function does.
- It is optional but highly recommended.
- It can be single-line or multiple-lines. It is common to use **triple quotes** which is capable of defining multi-line string.

Add Docstring to Module and Function

Modify the `myhello.py` file by adding Docstring for both module and functions.

- Module docstring must start at 1st line of the file.
- Function docstring must start immediately afte `def` statement

In [15]:
```
'''
My first module in Python.
It contains a few classic hello-world functions.
'''

def hello_once(name):
    '''Say hello once

    Args:
        A string containing the name
    '''
    print('Hello again {}'.format(name))

def hello_many(items):
    '''Say hello multiple times

    Args:
        A list of names
    '''
    for item in items:
        hello_once(item)

import sys
if __name__ == '__main__':
      print("Script name: {}".format(sys.argv[0]))
      if len(sys.argv) == 2:
              hello_once(sys.argv[1])
      if len(sys.argv) > 2:
              hello_many(sys.argv[1:])
```

```
Script name: C:\Users\User\Anaconda3\lib\site-packages\ipykernel_launcher.py
Hello again -f
Hello again C:\Users\User\AppData\Roaming\jupyter\runtime\kernel-3b1e6e2b-c17d-4d6b-8541-f6e437801503.json
```

How to access Docstring?

Docstring of a function or class can be accessed using **help()** function or `__doc__` attribute of the function or class.

- Start a Python REPL
- Import `myhello` module

```
$ import myhello
```

- Print docstring of module and function

```
$ print(myhello.__doc__)
$ help(myhello)
$ print(myhello.hello_many.__doc__)
$ help(myhello.hello_many)
```

6. Function Arguments and Returned Value

Input Parameters vs Arguments

A function may have 0 or more **input parameters**.
The input values passed into a function is commonly called **Arguments**.

There are different types of arguments:

- Default arguments
- Required arguments
- Keyword arguments
- Variable number of arguments

Let's define a funtion `simple_add` which takes in a few values and return sum of them.

```
def simple_add(a, b, c):
    s = a + b + c
    return s
```

In [16]:
```
def simple_add(a, b, c):
    s = a + b + c
    return s
```

Required Arguments

All arguments, `a` and `b` and `c` are **required arguments**. You need to pass in all required values before you can call `simple_add` function.

In [17]:
```
simple_add(1, 2, 3)
```

Out[17]:
```
6
```

Default Arguments

Default arguments have arguments with default values. When there is no value is passed, that argument will use its default value.

Modify the `simple_add` function to provide make `b` and `c` a default arguments.

```
def simple_add(a, b = 10, c = 20):
    s = a + b + c
    return s
```

Question: What if you only assign default value for `b` , i.e. only `b` is a default arguments?

Note: All required arguments must be before default arguments.

In [30]:
```
def simple_add(a, b = 10, c = 20):
    s = a + b + c
    return s

simple_add(20)
simple_add(20, c=15)
```

Out[30]:
```
45
```

Keyword Arguments

Instead of pass arguments in order, you can passs arguments identified by their name, i.e. keyword arguments.

- With keyword argument, order of arguments is not required.
- It can make your code easier to read.

In [19]:
```
simple_add(c = 30, a = 10, b = 20)

help(simple_add)
```

```
Help on function simple_add in module __main__:

simple_add(a, b=10, c=20)
```

Question: Can I omit some default arguments while using keyword arguments?

In [20]:
```
simple_add(a = 10, c = 5)
```

Out[20]:
```
25
```

Keyword Arguments using Dictionary

We can pack all keywoard arguments in one dictionary before pass it to a function.

In [21]:
```
def simple_add(a, b=20, c=30):
    s = a + b + c
    return s

myval = {'a':10, 'b':20, 'c':30}
print(simple_add(**myval))
myval = {'a':10, 'c':30}
print(simple_add(**myval))
```

```
60
60
```

Variable Number of Arguments

Sometimes you are not sure the exact number of arguments to be passed to a function. You can capture any number of argument with `*args` .

The asterisk `(*)` prefixing an variable name, e.g. `*args` , indicates that variable will be **unpacked** into multiple values.

- The variable holds values of all **nonkeyword variable arguments**.

Define another function add_all() which add all argument values and return result.

```
def add_all(*args):
    s = 0
    for i in args:
        s = s + i
    return s
```

Question: Can I replace `*args` with another name, e.g. `*inputs` ?

In [22]:
```
def add_all(*args):
    print(args)
    s = 0
    for i in args:
        s = s + i
    return s

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

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

Out[22]:
```
10
```

Variable Number of Keyword Arguments

KWargs or “keyword arguments” allows you to pass the keyworded, dictionary as arguments. Because dictionaries are almost always super useful.

```
    def print_values(**kwargs):
        for key, value in kwargs.items():
            print("{} is {}".format(key, value))

    print_values(my_name="ah boy", your_name="ah girl")
```

In [23]:
```
def print_values(**kwargs):
    for key, value in kwargs.items():
        print("{} is {}".format(key, value))

print_values(my_name="ah boy", your_name="ah girl")
```

```
my_name is ah boy
your_name is ah girl
```

Return Statement

A funciton may return one or more values implicitly or explicitly. To return value(s) out of a function, use `return` statement.

In [24]:
```
def simple_add(a, b):
    return a + b

result = simple_add(1,2)
print(result)
```

```
3
```

Return Multiple Values

A function may return **multiple** values. When multiple values are returned, they are packed into a tuple.

In [25]:
```
def simple_math(a, b):
    return a + b, a - b, a * b, a / b

result = simple_math(20,10)
print(result)
print(type(result))
```

```
(30, 10, 200, 2.0)
<class 'tuple'>
```

Use * to Grab Excess Items

The * can be used to grab multple values, e.g. indicate a function parameter can take in multiple arguments. It can also be used to grab multiple items in returned values.

In [26]:
```
result, *others = simple_math(20,10)
print(result)
print(others)
```

```
30
[10, 200, 2.0]
```

Implicit Return

If a function has no `return` statement in a function, or its `return` statement doesn't followed by any object, the function returns a `None` .

In [27]:
```
def fun1():
    print('before return')
    return
    print('after return')

def fun2():
    print('no return')

r1 = fun1()
r2 = fun2()
print('{} {}'.format(r1, r2))
```

```
before return
no return
None None
```

7.Variable Scope - Global vs Local

The scope of a variable determines the portion of the program where you can access a particular variable. There are two basic variable scopes in Python

- **Global variables:** variables defined **outside** a function body
- **Local variables:** variables defined **inside** a function body

Global and Local variables are in different scopes

- Local variables can be accessed only inside the function in which they are declared
- Global variables can be accessed throughout the program body

In [28]:
```
def myfunc():
    x = 10
    if x > 0:
        y = True
    else:
        y = False
    print(y)
    for item in range(5):
        z = item

    print(z)

myfunc()
print(x, y, z)
```

```
True
4

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-28-c6f0e743e7b6> in <module>
     12
     13 myfunc()
---> 14 print(x, y, z)

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

Accessing Global Variable

In this example, global variable `val` can be read both inside and outside the function.

```
val = 3;

def show():
    print("In function: val = {}".format(val));

show()
print("Outside function: val = {}".format(val));
```

In [ ]:
```
val = 3;

def show():
    print("In function: val = {}".format(val));

show()
print("Outside function: val = {}".format(val));
```

Accessing Local Variable

The `val` variable is a local variable. It does not exist outside the function.

- We use globals() to remove any `val` variable from previous example

```
# Clean up any `val` variable from previous example
if 'val' in globals():
    del val

def show():
    val = 3;
    print("In function: val = {}".format(val));

show()
print("Outside function: val = {}".format(val));
```

In [ ]:
```
# Clean up any `val` variable from previous example
if 'val' in globals():
    del val

def show():
    val = 3;
    print("In function: val = {}".format(val));

show()
print("Outside function: val = {}".format(val));
```

In [ ]:
```
def fun1():
    x = 10
    print(locals())

def fun2():
    y = 20
    print(locals())

fun1()
fun2()

if 'x' in locals():
    print('x')
if 'y' in locals():
    print('y')
if 'x' in globals():
    print('xg')
if 'y' in globals():
    print('yg')
```

Variable Created in Different Scope

Whenever a variable is assigned, it will be automatically created if it does not exist in current scope.

Question: Why there is an error in following code?

```
x = "global"

def foo():
    x = x * 2
    print(x)
foo()
```

Both global and local scopes have a `x` variable. In the function, x is used before initialized.

In [31]:
```
x = "global"

def foo():
    x = x * 2
    print(x)

foo()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-31-529895da442c> in <module>
      5     print(x)
      6
----> 7 foo()
      8 print(x)

<ipython-input-31-529895da442c> in foo()
      2
      3 def foo():
----> 4     x = x * 2
      5     print(x)
      6

UnboundLocalError: local variable 'x' referenced before assignment
```

In [ ]:
```
z = 2
print(id(z))
def foo():
    z = 3
    print(id(z))
    print(z)

foo()
print(z)
```

In following example, global variables `list1` and `list2` are pointing to same object. Inside the function, a local variable `list1` is created during assignment operation.

Global `list1` and local `list1` are different variables!

```
list1 = [1, 2]
list2 = list1
print(list1 is list2)
print(list1 == list2)

def myfun():
    print("Inside function")
    list1 = [3, 4]
    print(list1 is list2)
    print(list1 == list2)

myfun()
print("Outside function")
print(list1 is list2)
print(list1 == list2)
```

In [32]:
```
list1 = [1, 2]
list2 = list1
print(list1 is list2)
print(list1 == list2)

def myfun():
    print("Inside function")
    list1 = [3, 4]
    print(list1 is list2)
    print(list1 == list2)

myfun()
print("Outside function")
print(list1 is list2)
print(list1 == list2)
```

```
True
True
Inside function
False
False
Outside function
True
True
```

Keyword `global`

But what if I would like to modify a global variable in the function?

To access global a variable in a function, you can use the `global` keyword.

```
list1 = [1, 2]
list2 = list1
print(list1 is list2)
print(list1 == list2)

def myfun():
    global list1
    print("Inside function")
    list1.append(3)
    print(list1 is list2)
    print(list1 == list2)

myfun()
print("Outside function")
print(list1 is list2)
print(list1 == list2)
```

In [33]:
```
list1 = [1, 2]
list2 = list1
print(list1 is list2)
print(list1 == list2)

def myfun():
    global list1
    print("Inside function")
    list1.append(3)
    print(list1 is list2)
    print(list1 == list2)

myfun()
print("Outside function")
print(list1 is list2)
print(list1 == list2)
```

```
True
True
Inside function
True
True
Outside function
True
True
```

8. Pass by Value or Reference

All parameters (arguments) in the Python are passed by reference, i.e. only reference value of argument is copied to function parameter.

It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

Question: In following code, does modifying of list0 inside the function affects value of list1 which is defined outside the function?

```
def extend_list(list0):
    list0.extend(list0)
    print("Inside function: {}".format(list0))

list1 = [1,2,3]
extend_list(list1)
print("Outside function: {}".format(list1))
```

In [34]:
```
def extend_list(list0):
    list0.extend(list0)
    print("Inside function: {}".format(list0))

list1 = [1,2,3]
extend_list(list1)
print("Outside function: {}".format(list1))
```

```
Inside function: [1, 2, 3, 1, 2, 3]
Outside function: [1, 2, 3, 1, 2, 3]
```

Actually, `list0` and `list1` are two variables. They point to same object in memory, i.e. the list object. Since they are different variables, either of them can be assigned to another object. In this case, the original object is not modified.

Question: Does pointing list0 to another object affects list1 value?

```
def extend_list(list0):
    list0 = [7,8,9]
    print("Inside function: {}".format(list0))

list1 = [1,2,3]
extend_list(list1)
print("Outside function: {}".format(list1))
```

In [35]:
```
def extend_list(list0):
    list0 = [7,8,9]
    print("Inside function: {}".format(list0))

list1 = [1,2,3]
extend_list(list1)
print("Outside function: {}".format(list1))
```

```
Inside function: [7, 8, 9]
Outside function: [1, 2, 3]
```

In [ ]:
```
a = 1
print(id(a))

def dup(b):
    print(id(b))
    b = 2
    b = 1
    print(id(b))

dup(a)
```

9. Lambda Function (Anonymous Functions)

A lambda function is a small anonymous function, i.e. a simple function with no name.

- Instead of declaring them with the standard `def` keyword, it is declared using `lambda` keyword.

Basic Syntax

```
lambda arguments : expression
```

- Value(s) between `lambda` and `:` are input arguments
- Expression after `:` is evaluated and returned

In [36]:
```
#Normal function
def times2x(a):
    return a *2

print(type(times2x))
print(times2x(5))

# Lambda function
times2 = lambda a : a *2
print(type(times2))
print(times2(5))
```

```
<class 'function'>
10
<class 'function'>
10
```

Multiple Arguments

- Lambda Function can take any number of arguments, but can only have one expression.

In [ ]:
```
add_all = lambda a, b, c : a + b + c
print(add_all(5, 6, 2))
```

When to use Lambda?

Lambda functions can be used when a nameless function is needed for a short period of time.

- Sorting of collection
- Mapping of collection
- Filtering of collection
- Reducing of collection

Example: How to **sort** a collection of integer (positive and negative) by their absolute values?

- The `key` argument accepts a function which is used to determine the sort order, e.g. len()

```
list1 = [15, 5, -10, 20, -30]
```

In [ ]:
```
list1 = "Hello World from Singapore".split(' ')
print(list1)
sorted(list1, key=len, reverse=True)
```

In [ ]:
```
list1 = [15, 5, -10, 20, -30]
sorted(list1, key=lambda x: abs(x), reverse=True)
```

Filter Function

The `filter()` method filters the given iterable with the help of a function that tests each element in the iterable to be true or not.

- An element will be returned if it is evaluated to be True
- Basic syntax

```
filter(function, iterable)
```

Example: Follwoing code filter for items which contains `'2'` .

In [38]:
```
list1 = ['a1', 'a2', 'a3', 'b1', 'b2', 'b3', 'c1', 'c2', 'c3']

def contain2(item):
    if '2' in item:
        return True
    else:
        return False

result = filter(contain2, list1)
print(result2)
```

```
['a2', 'b2', 'c2']
```

Challenge: Can you convert above code using lambda?

In [39]:
```
list1 = ['a1', 'a2', 'a3', 'b1', 'b2', 'b3', 'c1', 'c2', 'c3']
result = filter(lambda x: '2' in x, list1)
print(list(result))
```

```
['a2', 'b2', 'c2']
```

Question: Without running following code, what items will be printed out?

In [ ]:
```
rlist1 = [1, -1, 'a', 0, False, True, '0', '', []]

filtered = filter(None, rlist1)
print(list(filtered))
```

Map Function

The `map()` function applies a function to each item in th list, and return result as item in new list.

- Basic syntax

  `map(function_to_apply, list_of_inputs)`

Example: Follwing example applies a maths equation `a + b*x` on list items.

In [40]:
```
list1 = [2, 3, 4, 5]

def trans(x):
    return 3 + 4*x

result = map(trans, list1)
print(list(result))
```

```
[11, 15, 19, 23]
```

Challenge: Convert above example using lambda function.

In [41]:
```
list1 = [2, 3, 4, 5]
result = map(lambda x: 3 + 4*x, list1)
print(list(result))
```

```
[11, 15, 19, 23]
```

Reduce Function

The `reduce()` function is a useful function for performing some computation on a list and returning result.

- It applies a rolling computation to sequential pairs of values in a list.
- It needs to be imported rom `functools` module.

- Basic syntax

```
reduce(func, sequence[, initial])
```

- When the initial value is missing, the function is called with first 2 items in the sequence.
- When the initial value is provided, the function is called with the initial value and the first item from the sequence.
- The `func` is a funtion which always takes in 2 values.

Example: Following example multiply all items together.

In [42]:
```
result = 1
list1 = [1, 2, 3, 4, 5]
for num in list1:
    result = result * num

print(result)
```

```
120
```

Use `reduce()` function instead of `for loop` .

In [44]:
```
import functools

list1 = [1, 2, 3, 4, 5]

def multiply(x, y):
    return x * y

result = functools.reduce(multiply, list1)
print(result)
```

```
120
```

Challenge: Convert above example using lambda function.

In [45]:
```
from functools import reduce
list1 = [1, 2, 3, 4, 5]
result = reduce((lambda x, y: x * y), list1, 1)
print(result)
```

```
120
```

Power of Lambda Functions

The power of lambda is better shown when you use them as an anonymous function inside another function.

Imaging you need to create 2 functions, which apply power of 2 and power of 3 on input argument respectively.

- You can create 2 separate functions.
- But what if you need functions from power 2 to power 10?

```
def mypower(n):
  return lambda a : a ** n

power2 = mypower(2)
power3 = mypower(3)
print(power2(10))
print(power3(10))
```

In [46]:
```
def mypower(n):
  return lambda a : a ** n

power2 = mypower(2)
power3 = mypower(3)
print(power2(10))
print(power3(10))
```

```
100
1000
```