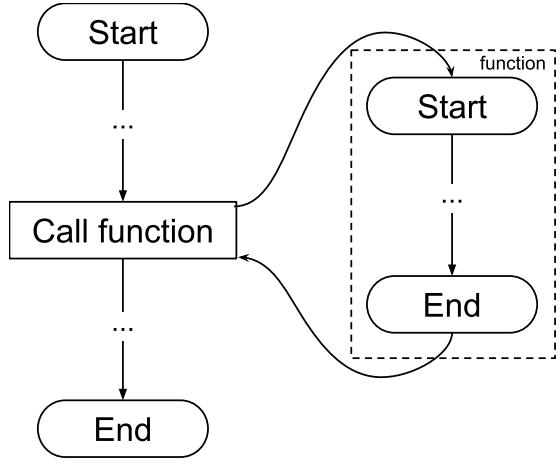
Python Workshop

6. Functions

Quick jump: Return | Scope | Parameters | Modules | Advanced topics

Concepts: Function

- **Function** allow us to define sequence of code to be re-used. It can also improve code readibility.
- This is like defining an operation to be used in a flow chart.




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

Explaining function

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

- The first line `def hello(name):` defines a function named `hello`, with one parameter `name`.
- Once again, the indentation defines the program block of the function.
- The line `print('Hello', name)` is the content of the function.

Executing function

- Code inside a function will not be executed until the function is called.
- To call a function, we add `( )` after the function name, and specify the values it needs.

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

hello('David')
```

Output:

```
Hello David
```

Recursion

- A function can call the function itself.
- In that case the function must have a terminal condition.

```
def listSum(myList):
    if len(myList) == 0:
        return 0

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

Self-learning topics (~90min)

- Function return
- Variable Scope
- Function parameters
- Modules
- Advanced topics:
  - Argument packing/unpacking
  - First-class function
  - Lambda function

Function return

Function return (1)

- A function can **return** a value for future use. For example:

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

- This define a function `add()` that accepts two parameters, `x` and `y`.
- The function return the result of `x + y`.

Function return (2)

- Returned value can be used immediately...

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

or be assigned to a variable.

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

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 be returned.

```
def func():
    pass

print(func())
```

Output:

```
None
```

Return value packing

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

```
def swap(a, b):
    return b, a
```

- Here, `b, a` is **packed** into a tuple, and returned.

Return value unpacking

- When using the previous function, the tuple will be **unpacked** automatically if we specify a list of variables:

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

Output:

```
20 10
```

Variable scope

Variable scope (1)

- Variable defined outside of a function is **global**.
- **global** variables can be accessed in a function.

```
x = 1
def func():
    print(x)
```

Variable scope (2)

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

```
def func():
    xxx = 1
    print(XXX)

func()
print(xxx)
```

- The last line will cause an error unless `xxx` is previously defined.

Variable scope (3)

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

```
x = 1
def func():
    print(x)
    x = 2
    print(x)

func()
```

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

Variable scope (4)

- If we need to assign a value to a global variable, we must declare the variable global in the function first:

```
x = 1
def func():
    global x
    print(x)
    x = 2
    print(x)

func()
```

- `global x` declares that the global `x` should be used.

Global or local?

- 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

```
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:

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

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

Output:

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

Passing a list

- When a list is passed into a function, the effect is the same as if assignment operator = is used. The code:

```
def f(myList2):
  mylist2[0] = 4

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

will have the same result as:

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

Default values (1)

- We can set a **default** for some of the parameters.
- In this way, the function can takes less values and uses the defaults.

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

hello()
hello('Hi')
hello('Hi', 'Jason')
hello('Hi', 'Jason', 'Welcome')
```

Default values (2)

- 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 (1)

- We can choose to specify a value by **keyword**:

```
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?')
```

- All three function calls produce the same result.

Keyword arguments (2)

- Once a keyword argument is specified, all remaining values must be specified as keyword arguments.
- So this is invalid:

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

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

Keyword and Defaults

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

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

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

Exercises (1)

Exercise 1

- 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.

Exercise 1 test cases

- You can always 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)))
```

Modules

Modules

- Another level of reuse is to reuse functions defined in another file.
- Suppose you have written the following function in a file named `helloUtil.py`.

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

- In another file, you can import this function by the statement:

```
from helloUtil import hello
hello()
```

Import *

- The statement `from XX import YY` specify that we import the name `YY` from file `XX.py`.
- Name `YY` can be any variable or function name.
- File `XX.py` will be retrived in the same folder of the code, or the library path.
- We can import all names with `from XX import *`.

```
from helloUtil import *
hello()
```

Name clashes

- Consider this example:

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

from helloUtil import hello
hello()
```

- The name `hello` is overwritten by import, and so the original definition is gone.

Import as

- To avoid the previous problem, we can use `as` to rename the imported function.

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

from helloUtil import hello as hello2
hello()
hello2()
```

Import module

- Using `import *` is convenient but it will import all names from the file which may never be used, which is not desirable.
- One possible solution is to import the file as a **module** instead.

```
import helloUtil
helloUtil.hello()
```

- Note that in this case, we need to access the functions from the module name instead.

Import module as

- Similarly, we can import a module and rename it.

```
import helloUtil as hello
hello.hello()
```

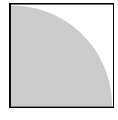
Standard libraries

- Python provide a lot of standard libraries that could be included this way. For example, `math` and `random`.
- You can read the corresponding references:
  - Math: https://docs.python.org/3/library/math.html
  - Random: https://docs.python.org/3/library/random.html

Exercises (2)

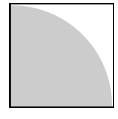
Monte Carlo method (1)

- The Monte Carlo method is one way of finding π using random values.
- Consider a square of size 1×1 with a quater of a circle with raduis 1 as shown below.



- By considering the area of the two regions in the figure, the chance of a random dot falling in the shaded region equals $\pi/4$.

Monte Carlo method (2)



- If we generate X random dots in the square, and counted that Y of them being in the circle, we can estimate that π equals $4Y/X$.

Pseudo code

- Here is the pseudo code of the Monte Carlo method.

```
count = 0
while number of random points < target number of points
    generate random value x in range [0, 1)
    generate random value y in range [0, 1)
    if x^2 + y^2 < 1
        increment count

pi = 4 * count / total number of random points
```

Exercise 2 task

- Write a function that implement the Monte Carlo method.
- The function should accept one argumnent, `N`, where `N` being the number of random points generated.
- The function should return the value of π calculated.
- Test your function with different values of `N`.

Advanced topics

- Remember: you may skip this section and come back later.

Argument packing and unpacking

Argument unpacking

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

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

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

Output:

```
Hi Jason
Welcome!
```

Argument unpacking usage

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

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

Output:

```
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.

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

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

Output:

```
hello Jason
Welcome!
```

Variable keyword arguments (1)

- 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:

```
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})
```

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

Variable keyword arguments (2)

- Output of the previous program:

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

Variable arguments (1)

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

```
def func(a, b, *c):
    print(a, b, c)

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

Output:

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

Variable arguments (2)

- 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 this:

```
def func(a, *b, c):
    print(a, b, c)
```

- `c` must be specified by keyword:

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

Variable arguments (3)

- `*name` and `**name` can be used together.
- `**name` must be placed at the end.

```
def func(*b, **c):
    print(b, c)

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

Output:

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

First-class functions

First-class functions

- Python functions are **first-class functions**, all function is treated as a variable.
    - This is a feature very commonly seen in modern programming languages.
- We can therefore assign a function to a variable:

```
def myFunc():
    print('This is myFunc')

myFunc2 = myFunc
myFunc2()
```

Namespace

- Function name and variable name uses the same 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:

```
def func():
    pass

func = 0
func()
```

Function as arguments

- Since function can be used as a variable, we can pass a function as an argument.

```
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))
```

- The above code will calculate and print the sum of 1^2 to 9^2, which equals `285`.

Local function

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

```
def func():
    def innerFunc(a):
        return a**2

    return innerFunc(10)

print(func())
```

Output:

```
100
```

Function as returned value

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

```
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))
```

Lambda function

Lambda function

- 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.

Lambda function (2)

- One of the previous example can be modified to:

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

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

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.

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