Functions, packages, and modules

- **Packages**, are collections of related useful things such as multiplying matrices, calculating a logarithm, or making a graph, and each package has a name by which you can refer to it.

- ***from math import log.*** This tells the computer to "import" the logarithm function from the math package.

- You need to import each function you use only once per program: once the function has been imported it continues to be available until the program ends. You can also import all the functions in a package with a statement of the form ***from math import *,*** but it is better not to do this, which might give many unexpected behaviors.

- Some large packages are split into smaller sub-packages, called modules, e.g. ***numpy.linalg*** and ***numpy.fft***

Functions

- **Functions** are a convenient way to put your codes into different blocks. The advantages of using functions are as following:

1. Make the codes easily reuse
2. Make the codes more readable
3. Make the codes shorter
4. Make it easy to organize our codes

- The basic form of function in Python

1. Start with the function name
2. Followed by parentheses () and a colon :
3. There are some spaces for the first line
4. Functions can receive arguments (variables passed to the functions from the callers), which are contained in parentheses between the function name and the colon.

Examples

- The simplest function

```
def function_name():
    1st block lines
    2nd block lines
    ...
```

```
def my_copyright():
    print("*****************************************")
    print("***  programmed by XXX for MSDM5002   ***")
    print("-----------------------------------------")
    print("***  You can use it as you like, but  ***")
    print("***  there might be many bugs. If you ***")
    print("***  find some bugs, please send them ***")
    print("***  to XXX@ust.hk.                   ***")
    print("*****************************************")
```

- The simplest function with arguments

```
def function_name(1st_argu, 2nd_argu, ...):
    1st block lines
    2nd block lines
    ...
```

```
def my_copyright2(date):
    print("*****************************************")
    print("***  programmed by XXX for MSDM5002   ***")
    print("***         date:",   date, "          ***")
    print("***-----------------------------------***")
    print("***  You can use it as you like, but  ***")
    print("***  there might be many bugs. If you ***")
    print("***  find some bugs, please send them ***")
    print("***  to XXX@ust.hk.                   ***")
    print("*****************************************")
```

Optional arguments and default parameters

- In Python, there can only be one function with a particular name defined in the scope. If you define two functions with the same name, the first one will be overwritten by the second one. However, sometimes, similar function might need to deal with different tasks. For example, we want to do the summation for two or three variables.
- In Python, we can use **optional arguments** to realize this. The way is to set some arguments with default values.

```
def my_sum(a,b,c=0):
    if (type(a)!=int and type(a)!=float) or (type(b)!=int and type(b)!=float):
        print("your input are not numbers")
        return
    s=a+b+c
    return s
a=1
b=3
c=10
s=my_sum(a,b);
print(a,'+',b,'=', s)
s2=my_sum(a,b,c);
print(a,'+',b,'+',c,'=', s2)
```

Use keywords to pass parameters

- In Python, besides the positional parameters, we can also use keywords to pass the parameters.
- The keyword parameters could be in any order – they don't have to match the order in the function definition.
- We can mix positional and keyword parameters, but the keyword parameters must come after any positional parameters.

```
def my_sum_scale(a,b,third=0,scaler=1):
    print('a=',a,'b=',b,'third=',third,'scaler=',scaler)
    s=(a+b**3+third)*scaler
    return s
print('my_sum_scale(1,2)=',my_sum_scale(1,2))
print('my_sum_scale(1,2,3)=',my_sum_scale(1,2,3))
print('my_sum_scale(1,2,3,4)=',my_sum_scale(1,2,3,4))
my_sum_scale(1,2,third=3,scaler=4)
my_sum_scale(1,2,scaler=4,third=3)
```

Return in functions

- The more common usage of a function is that we use a function to do some calculations and then we can get the result. To get the results from the calculations in a function, we need "return" statement.

```
def function_name(1st_argu, 2nd_argu, ...):
    1st block lines
    2nd block lines
    ...
    return var1, var2, ...
```

```
def my_sum(a,b):
    s=a+b
    return s
a=1
b=3
s=my_sum(a,b);
print(a,'+',b,'=', s)
```

```
def my_sum_product(a,b):
    s=a+b
    p=a*b
    return s,p

a=1
b=3

s,p=my_sum_product(a,b)
print(a,'+',b,'=', s)
print(a,'*',b,'=', p)
```

Additional usage of return

- All functions do return something, even if we don't define a return value – the default return value is None.

```
In [99]: a=my_copyright();
****************************************
***  programmed by XXX for PHYS6810I  ***
----------------------------------------
***  You can use it as you like, but  ***
***  there might be many bugs. If you ***
***  find some bugs, please send them ***
***  to XXX@ust.hk.                   ***
****************************************

In [100]: print(a)
None
```

- Moreover, when a **return** statement is reached, the flow of control immediately exits the function – any further statements in the function body will be skipped. We can sometimes use this to reduce the number of conditional statements we need to use inside a function:

```
def my_sum(a,b):
    if (type(a)!=int and type(a)!=float) or (type(b)!=int and type(b)!=float):
        print("your input are not numbers")
        return
    s=a+b
    return s
a=1
b=3
s=my_sum(a,b);
print(a,'+',b,'=', s)
```

Lifetime of a variable

- Lifetime is the time duration where an object/variable is in a valid state. In other words, an object/variable can be only accessed by the codes from the memory before the life ends, otherwise it is not valid.

```
def my_sum_test(a,b): ##A4, A5
    c=a+b  #A6
    print("c in the function: ",c) #A6
    a=100 #A4
    print("a in the function: ",a) #A4
    print("d in the function: ",d) #A0
    # d=10  ## can we define it here?? #A7
    return c #A6

d=1000 #A0
a=1  #A1
b=3  #A2
c=10 #A3

print("a in main code: ",a) #A1
my_sum_test(a,b)

print("d in main code: ",d) #A0

print("c in the main code: ", c) #A3
print("a in main code after calling the function: ",a) #A1
```

Functions with same name – bad example

- A function can be defined at an arbitrary position of the codes, and you can define two different functions with the same name (but **DO NOT DO IT**).

```
def my_sum(a,b):
    s=a+b
    return s
a='a'
b='b'
s=my_sum(a,b);
print(a,'+',b,'=', s)

def my_sum(a,b):
    if (type(a)!=int and type(a)!=float) or (type(b)!=int and type(b)!=float):
        print("your input are not numbers")
        return
    s=a+b
    return s
a=1
b=3
s=my_sum(a,b);
print(a,'+',b,'=', s)
```

Recursive function

- We can make a function call itself, which is known as a recursive function.
- Recursive function is allowed in Python. Sometimes, it will make the code very beautiful and elegant by using recursive functions, while it will also make the code very hard to read and understand. It is better NOT to use recursive function.

```
def my_recursive_function(n):
    if n == 0:
        return 0

    if n == 1:
        return 1

    return my_recursive_function(n - 1) + my_recursive_function(n - 2)

for ni in range(10):
    print(my_recursive_function(ni))
```

- It is a function to calculate Fibonacci sequence.

Note about recursive functions

- There must be conditions which will allow it to stop recursion – an end case in which the function doesn't call itself.
- It is very difficult to write a fail-safe recursive function. For example, it is very hard to write some codes to exclude that we could input -1 for the Fabonacci function.
- ***Any*** recursive function can be re-written in an ***iterative*** way which avoids recursion.
- Try not use recursive functions even sometimes it seems to be very natural or much better to use them.

Practice

- Write the Fibonacci function in an ***iterative*** way which avoids recursion.

```
def fibonacci(n):
    current, nextone = 0, 1

    for i in range(n):
        current, nextone = nextone, current + nextone

    return current

for ni in range(10):
    print(fibonacci(ni))
```

Documentation Strings

- The first line should always be a short, concise summary of the object's purpose. For brevity, it should not explicitly state the object's name or type, since these are available by other means (except if the name happens to be a verb describing a function's operation). This line should begin with a capital letter and end with a period.
- If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print(my_function.__doc__)
Do nothing, but document it.

    No, really, it doesn't do anything.
```

Build-in functions

		Built-in Functions		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

- https://docs.python.org/3/library/functions.html

Object-oriented programming

- Python is an **object-oriented programming** language. It means that we can also define classes and use objects to represent all the concepts we use in real world.

- Object-oriented (面向对象) programming (OOP) is based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. Significant object-oriented languages include **Java, C++, C#, Python, PHP, JavaScript**, etc.

- Procedural (面向过程) programming is derived from structured programming and based upon the concept of the procedure call. Procedures, also known as routines, subroutines, or functions, simply contain a series of computational steps to be carried out. **Fortran** and **C** are the most significant sample.

Class and object in Python

- Object (对象) is an encapsulation of different variables and functions into a single entity.
- Class (类) is a template to create the objects. Object gets its variables and functions from the class.
- We can use many collection ways to group related data together, and we can use functions to create shortcuts for commonly used groups of statements.
- A function performs an action using some sets of input parameters, but not all functions are applicable to all kind of data. Classes are a way of grouping together related data and functions which act upon that data.
- You can think a class as a kind of data type, just like string or integer. When you create an object of that data type, you make an ***instance (实列)*** of a class.
- **In Python, everything is an object**. Even classes and types are themselves objects, and they are of type ***type***. You can find out the type of any object using type().

Attributes and methods for objects

- The data values which we store inside an object are called **attributes(属性)**, and the functions which are associated with the object are called **methods(方法)**. We have already used the methods of some built-in objects, like strings and lists.

- When we design our own objects, we have to decide how we are going to group things together, and what our objects are going to represent.

- Sometimes we write objects which map very intuitively onto things in the real world. However, it isn't always necessary, desirable or even possible to make all code objects perfectly analogous to their real-world counterparts. Sometimes we may create objects which don't have any kind of real-world equivalent, just because it's useful to group certain functions together.

An example

```
import datetime # we will use this for date objects

class Person:

    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        self.surname = surname
        self.birthdate = birthdate

        self.address = address
        self.telephone = telephone
        self.email = email

    def age(self):
        today = datetime.date.today()
        age = today.year - self.birthdate.year

        if today < datetime.date(today.year, self.birthdate.month, self.birthdate.day):
            age -= 1

        return age

person = Person(
    "Jane",
    "Doe",
    datetime.date(1992, 3, 12), # year, month, day
    "No. 12 Short Street, Greenville",
    "555 456 0987",
    "jane.doe@example.com"
)
print(type(person))
print(person.name)
print(person.email)
print(person.age())
```

Instance attributes

- It is important to note that the attributes set on the object in the ***__init__ function*** do NOT form an exhaustive list of all the attributes that our object is ever allowed to have.
- In Python, you can add new attributes, and even new methods, to an object on the fly. In fact, there is nothing special about the __init__ function when it comes to setting attributes.
- You can even add a completely unrelated attribute from outside the object, but it is a bad practice. **DO NOT DO IT.**
- We can use ***getattr(), setattr()*** and ***hasattr()*** to access, change and check the attributes.
- There are several ways to get all the attribute of an object, **dir(), vars(), __dir__(),__dict__**

Class attributes

- We can also define attributes which are set on the class. **These attributes will be shared by all instances of that class.** In many ways they behave just like instance attributes, but there are some differences that you should be aware of.
- We define class attributes in the body of a class, at the same indentation level as method definitions (one level up from the insides of methods):

```
class Person:

    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms')

    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        self.surname = surname
        self.birthdate = birthdate
```

- Class attributes are often used to define constants which are closely associated with a particular class.
- Class attributes can also sometimes be used to provide default attribute values.

comparison

- We should be very careful when a class attribute is of a mutable type – because if we modify it in-place, we will affect all objects of that class at the same time. Remember that all instances share the same class attributes:

```python
class Person:
    pets = []

    def add_pet(self, pet):
        self.pets.append(pet)

jane = Person()
bob = Person()

jane.add_pet("cat")
bob.add_pet("dog")
print(jane.pets)
print(bob.pets) # oops!
```

```python
class Person:

    def __init__(self):
        self.pets = []

    def add_pet(self, pet):
        self.pets.append(pet)

jane = Person()
bob = Person()

jane.add_pet("cat")
bob.add_pet("dog")
print(jane.pets)
print(bob.pets)
```

A good example of usage of class attributes

- To use class attributes to limit all the possible values for the instance attributes

```python
class Person:
    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms')

    def __init__(self, title, name, surname, allowed_titles=TITLES):
        if title not in allowed_titles:
            raise ValueError("%s is not a valid title." % title)

        self.title = title
        self.name = name
        self.surname = surname
```

Inheritance

- Python also supports inheritance. The syntax for a derived class definition looks like this.

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

- The name BaseClassName must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed.

```
class DerivedClassName(modname.BaseClassName):
```

- Python supports a form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Prevent creating new attributes outside __init__

```
class FrozenClass(object):
    __isfrozen = False
    def __setattr__(self, key, value):
        if self.__isfrozen and not hasattr(self, key):
            raise TypeError( "%r is a frozen class" % self )
        object.__setattr__(self, key, value)

    def _freeze(self):
        self.__isfrozen = True

class Person(FrozenClass):
    def __init__(self):
        self.name=''
        self.title=''

        self._freeze() # no new attributes after this point.
```

Modules

- All software projects start out small, and you are likely to start off by writing all your codes in a single file. As your project grows, it will become increasingly inconvenient to do this.
- At some point it will be a good idea to tidy up the project by splitting it up into several files, putting related classes or functions together in the same file.
- Python provides a mechanism for creating a ***module*** from each file of source code. You can use code which is defined inside a module by importing the module using the ***import*** keyword – we have already done this with some built-in modules in previous examples.
- Each module has its own namespace, so it’s OK to have two classes which have the same name, as long as they are in different modules. If we import the whole module, we can access properties defined inside that module with the **.** operator:

import

- We can import the whole module by **import module_name**
- We can import the whole module and give it a new name by **import module_name as new_name**

- We can also import specific classes or functions from the module by **from module_name import function_name**

- We can also import specific classes or functions from the module and give it a new name by **from module_name import function_name as new_function_name**

```python
import datetime
today = datetime.date.today()

import datetime as dt
today = dt.date.today()

from datetime import date
today = date.today()

from datetime import date as dtt
today = dtt.today()
```

Import your own functions

- We can also import our own functions in other files

from file_name import function_name

- The easiest way is to put all the files in the same folder

```
from All_Copy_right import *



my_copyright()
my_copyright2('10/09/2021')
my_copyright3('J. Liu', 'liuj@ust.hk', '10/09/2021')
my_copyright4('J Liu', 'liuj@ust.hk', '10/09/2021')
```

Create module

- Creating a module is as simple as writing code into a file.
- A module which has the same name as the file (without the .py suffix) will automatically be created.
- You will be able to import it if you run Python from the directory where the file is stored, or a script which is in the same directory as the other Python files.
- If you want to be able to import your modules no matter where you run Python, you should package your code and install it.
- You can install the module by writing a setup.py file, which is the specification for the module/packages, and then use the following command to install it

Python3 setup.py install

- If everything has gone well, you should be able to import the module from anywhere on your system.

Packages

- Just as a module is a collection of classes or functions, a ***package*** is a collection of modules. We can organize several module files into a directory structure.

- There are various tools which can then convert the directory into a special format (also called a “package”) which we can use to install the code cleanly on our computer (or other people’s computers). This is called ***packaging***.

- A library called ***Distribute*** is currently the most popular tool for creating Python packages, and is recommended for use with Python 3. It isn’t a built-in library, but can be installed with a tool like ***pip*** or ***easy_install***.

Useful modules in the Standard Library

- Date and time: datetime

Datetime.datime.today(), Datetime.datime.today().year, etc.

- Mathematical functions: math, cmath

math.pi, math.e, math.lot(), math.sqrt(), math.sin(), etc.

- Pseudo-random numbers: random

random.random(), random.seed(10), etc.

- Base N-dimensional array package: NumPy
- Fundamental library for scientific computing: SciPy library
- Comprehensive 2D plotting: Matplotlib
- Symbolic mathematics: Sympy
- Data and Machine learning: sklearn, pandas, tensorflow, etc.

Referred websites

- https://python-textbok.readthedocs.io/en/1.0/Introduction.html
- https://wiki.python.org/moin/UsefulModules#Plotting
- https://www.scipy.org/
- http://www.numpy.org/
- http://pandas.pydata.org/
- https://www.tensorflow.org/
- http://scikit-learn.org/
- https://docs.python.org/3/library/index.html