

# MY JOURNAL TO PYTHON

4110E233 - 高瑞夫

# AGENDA

1

LEARNING PYTHON

2

INPUT AND OUTPUT

3

DATA TYPES

4

OPERATORS

5

CONTROLS

6

LOOPS

7

FUNCTION

# LEARNING PYTHON

# CONTENT

- ▶ Python Introduction
  - ▶ What is Python?
  - ▶ What can Python do?
  - ▶ Why Python?
  - ▶ Good to know
  - ▶ Python Syntax compared to other programming languages
  - ▶ Python Jobs
  - ▶ Why to Learn Python?
  - ▶ Python Online Interpreter
    - ▶ One
    - ▶ Two

# CONTENT

- ▶ Python Syntax
- ▶ Python Comments
- ▶ Python Variables
  - ▶ Variable Names
  - ▶ Assign Multiple Values
  - ▶ Output Variables
  - ▶ Global Variables

# LEARNING PYTHON

Introduction

# What is Python?

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

- ▶ web development (server-side),
- ▶ software development,
- ▶ mathematics,
- ▶ system scripting.

# What can Python do?

- ▶ Python can be used on a server to create web applications.
- ▶ Python can be used alongside software to create workflows.
- ▶ Python can connect to database systems. It can also read and modify files.
- ▶ Python can be used to handle big data and perform complex mathematics.
- ▶ Python can be used for rapid prototyping, or for production-ready software development.



# Why Python?

- ▶ Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- ▶ Python has a simple syntax similar to the English language.
- ▶ Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- ▶ Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- ▶ Python can be treated in a procedural way, an object-oriented way or a functional way.

# Good to know

- ▶ The most recent major version of Python is Python 3, which we shall be using in this tutorial. However, Python 2, although not being updated with anything other than security updates, is still quite popular.
- ▶ In this tutorial Python will be written in a text editor. It is possible to write Python in an Integrated Development Environment, such as Thonny, Pycharm, Netbeans or Eclipse which are particularly useful when managing larger collections of Python files.

# Python Syntax compared to other programming languages

- ▶ Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- ▶ Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- ▶ Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

# Python Jobs

Python is very high in demand and all the major companies are looking for great Python Programmers to develop websites, software components, and applications or to work with Data Science, AI, and ML technologies.

Today a Python Programmer with 3-5 years of experience is asking for around \$150,000 annual package and this is the most demanding programming language in America. Though it can vary depending on the location of the Job. It's impossible to list all of the companies using Python, to name a few big companies are Google, Intel, NASA, PayPal, IBM, and many more...

# Why to Learn Python?

Python is consistently rated as one of the world's most popular programming languages. Python is fairly easy to learn, so if you are starting to learn any programming language then Python could be your great choice. Today various Schools, Colleges and Universities are teaching Python as their primary programming language. There are many other good reasons which makes Python as the top choice of any programmer:

- ▶ Python is Open Source which means its available free of cost.
- ▶ Python is simple and so easy to learn
- ▶ Python is versatile and can be used to create many different things.
- ▶ Python has powerful development libraries include AI, ML etc.
- ▶ Python is much in demand and ensures high salary

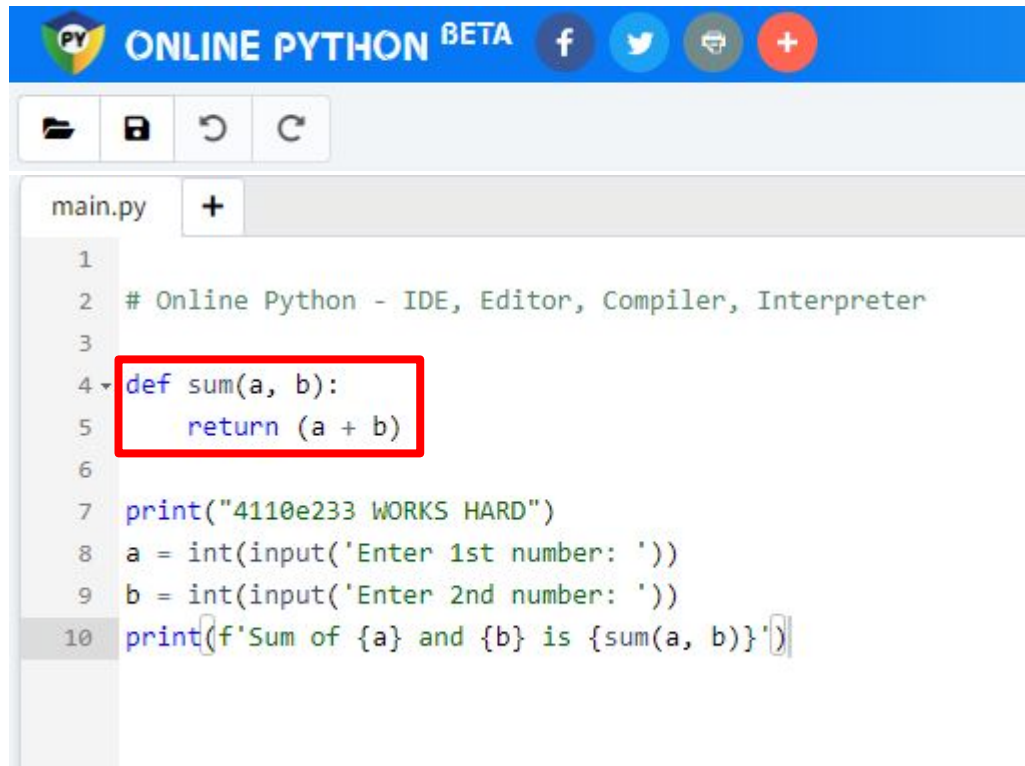
# Python Online Interpreter: One

  
Python Online  
Compiler

Interactive Python  
Course

main.py	Shell
<pre>1 # Online Python compiler (interpreter) to run   Python online. 2 # Write Python 3 code in this online editor and   run it. 3 print("4110e233 WORKS HARD") 4 print("Hello world")</pre>	<pre>4110e233 WORKS HARD Hello world &gt; 3**5 243 &gt;  </pre>

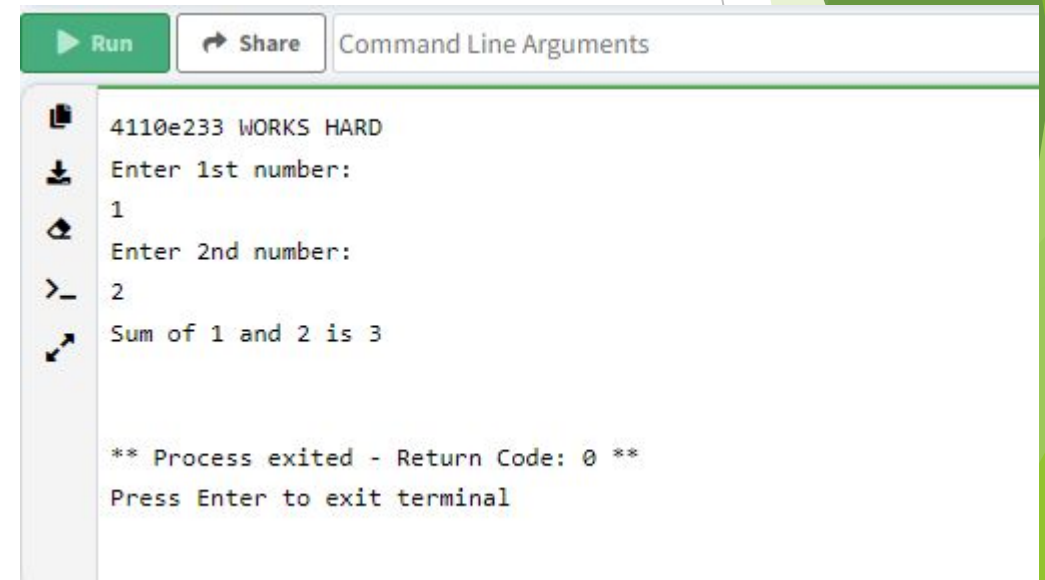
# Python Online Interpreter: Two



The screenshot shows the 'ONLINE PYTHON BETA' web interface. At the top, there's a blue header with the logo and social media icons. Below it is a toolbar with icons for file operations. The main area is a code editor with a file named 'main.py'. The code is as follows:

```
1
2 # Online Python - IDE, Editor, Compiler, Interpreter
3
4 def sum(a, b):
5     return (a + b)
6
7 print("4110e233 WORKS HARD")
8 a = int(input('Enter 1st number: '))
9 b = int(input('Enter 2nd number: '))
10 print(f'Sum of {a} and {b} is {sum(a, b)}')
```

The function definition on lines 4 and 5 is highlighted with a red rectangle.



The screenshot shows the execution output of the Python script. The interface includes 'Run' and 'Share' buttons, and a 'Command Line Arguments' field. The output is displayed in a terminal window with the following content:

```
4110e233 WORKS HARD
Enter 1st number:
1
Enter 2nd number:
2
Sum of 1 and 2 is 3

** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

# LEARNING PYTHON

Syntax



# Python Indentation

- ▶ Indentation refers to the spaces at the beginning of a code line.
- ▶ Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.
- ▶ Python uses indentation to indicate a block of code.

```
print("4110E233 - 高瑞夫")  
if 3 > 2:  
    print("Three is greater than two!")
```

```
4110E233 - 高瑞夫  
Three is greater than two!
```

Correct Indentation

```
print("4110E233 - 高瑞夫")  
if 3 > 2:  
print("Three is greater than two!")
```

```
Traceback (most recent call last):  
  File "/usr/lib/python3.8/py_compile.py", line 144, in compile  
    code = loader.source_to_code(source_bytes, dfile or file,  
  File "<frozen importlib._bootstrap_external>", line 846, in source_to_code  
  File "<frozen importlib._bootstrap>", line 219, in _call_with_frames_removed  
  File "./prog.py", line 3  
    print("Three is greater than two!")  
    ^  
IndentationError: expected an indented block  
  
During handling of the above exception, another exception occurred:  
  
Traceback (most recent call last):  
  File "<string>", line 1, in <module>  
  File "/usr/lib/python3.8/py_compile.py", line 150, in compile  
    raise py_exc  
py_compile.PyCompileError: Sorry: IndentationError: expected an indented block (prog.py, line 3)
```

Incorrect Indentation

# LEARNING PYTHON

Comments

# Creating a Comment

- Comments starts with a `#`, and Python will ignore them

```
print("4110E233 - 高瑞夫")  
#This is a comment.  
print("Hello, Universe!")
```

```
4110E233 - 高瑞夫  
Hello, Universe!
```

- Comments can be placed at the end of a line, and Python will ignore the rest of the line:

```
print("4110E233 - 高瑞夫")  
print("Hello, Universe!") #This is a comment.
```

```
4110E233 - 高瑞夫  
Hello, Universe!
```

- A comment does not have to be text that explains the code, it can also be used to prevent Python from executing code:

```
print("4110E233 - 高瑞夫")  
#print("Hello, Universe!")
```

```
4110E233 - 高瑞夫
```

# LEARNING PYTHON

Variables

# Variables

Variables are containers for storing data values.

- Creating Variables

Python has no command for declaring a variable.

A variable is created the moment you first assign a value to it.

```
print("4110E233 - 高瑞夫")  
x = 10  
y = "reyb"  
print(x)  
print(y)
```

```
4110E233 - 高瑞夫  
10  
reyb
```

## ● Creating Variables

Variables do not need to be declared with any particular *type*, and can even change type after they have been set.

```
print("4110E233 - 高瑞夫")  
x = 10          # x is of type int  
x = "Lyle"      # x is now of type str  
print(x)
```

```
4110E233 - 高瑞夫  
Lyle
```

## ● Casting

If you want to specify the data type of a variable, this can be done with casting.

```
print("4110E233 - 高瑞夫")  
  
x = str(10)  
y = int(10)  
z = float(10)  
  
print(x)  
print(y)  
print(z)
```

```
4110E233 - 高瑞夫  
10  
10  
10.0
```

## ● Get the Type

You can get the data type of a variable with the `type()` function.

```
print("4110E233 - 高瑞夫")  
x = 10  
y = "Rhonny"  
print(type(x))  
print(type(y))
```

```
4110E233 - 高瑞夫  
<class 'int'>  
<class 'str'>
```



- Single or Double Quotes?

String variables can be declared either by using single or double quotes:

```
print("4110E233 - 高瑞夫")  
x = "Reyb"  
print(x)  
#double quotes are the same as single quotes:  
x = 'Reyb'  
print(x)
```

```
4110E233 - 高瑞夫  
Reyb  
Reyb
```

- Case-Sensitive

Variable names are case-sensitive.

```
print("4110E233 - 高瑞夫")  
  
a = 10  
A = "Karen"  
  
print(a)  
print(A)
```

```
4110E233 - 高瑞夫  
10  
Karen
```

# Variable Names

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total\_volume). Rules for Python variables:

- ▶ A variable name must start with a letter or the underscore character
- ▶ A variable name cannot start with a number
- ▶ A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and \_ )
- ▶ Variable names are case-sensitive (age, Age and AGE are three different variables)

## Remember that variable names are case-sensitive

### Legal variable names

```
print("4110E233 - 高瑞夫")
```

```
myvar = "Loonny"  
my_var = "Loonny"  
_my_var = "Loonny"  
myVar = "Loonny"  
MYVAR = "Loonny"  
myvar2 = "Loonny"
```

```
print(myvar)  
print(my_var)  
print(_my_var)  
print(myVar)  
print(MYVAR)  
print(myvar2)
```

```
4110E233 - 高瑞夫  
Loonny  
Loonny  
Loonny  
Loonny  
Loonny  
Loonny
```

### Illegal variable names

```
print("4110E233 - 高瑞夫")
```

```
2myvar = "Loonny"  
my-var = "Loonny"  
my var = "Loonny"
```

#This example will produce an error in the result

```
Traceback (most recent call last):  
  File "/usr/lib/python3.8/py_compile.py", line 144, in compile  
    code = loader.source_to_code(source_bytes, dfile or file,  
  File "<frozen importlib._bootstrap_external>", line 846, in source_to_code  
  File "<frozen importlib._bootstrap>", line 219, in _call_with_frames_removed  
  File "./prog.py", line 3  
    2myvar = "Loonny"  
    ^  
SyntaxError: invalid syntax  
  
During handling of the above exception, another exception occurred:  
  
Traceback (most recent call last):  
  File "<string>", line 1, in <module>  
  File "/usr/lib/python3.8/py_compile.py", line 150, in compile  
    raise py_exc  
py_compile.PyCompileError:  File "./prog.py", line 3  
    2myvar = "Loonny"  
    ^  
SyntaxError: invalid syntax
```

# Multi Words Variable Names

Variable names with more than one word can be difficult to read.

There are several techniques you can use to make them more readable:

- Camel Case

Each word, except the first, starts with a capital letter:

```
print("4110E233 - 高瑞夫")  
myVariableName = "Lullu"  
print(myVariableName)
```

```
4110E233 - 高瑞夫  
Lullu
```

- Pascal Case

Each word starts with a capital letter:

```
print("4110E233 - 高瑞夫")  
MyVariableName = "Lullu"  
print(MyVariableName)
```

```
4110E233 - 高瑞夫  
Lullu
```

- Snake Case

Each word is separated by an underscore character:

```
print("4110E233 - 高瑞夫")  
my_variable_name = "Lullu"  
print(my_variable_name)
```

```
4110E233 - 高瑞夫  
Lullu
```

# Assign Multiple Values

- Many Values to Multiple Variables

Python allows you to assign values to multiple variables in one line:

```
print("4110E233 - 高瑞夫")  
  
x, y, z = "Apple", "Mango", "Strawberry"  
  
print(x)  
print(y)  
print(z)
```

```
4110E233 - 高瑞夫  
Apple  
Mango  
Strawberry
```

**Note:** Make sure the number of variables matches the number of values, or else you will get an error.



## ● One Value to Multiple Variables

And you can assign the *same* value to multiple variables in one line:

```
print("4110E233 - 高瑞夫")  
  
x = y = z = "Apple"  
  
print(x)  
print(y)  
print(z)
```

```
4110E233 - 高瑞夫  
Apple  
Apple  
Apple
```

## ● Unpack a Collection

If you have a collection of values in a list, tuple etc. Python allows you to extract the values into variables. This is called *unpacking*.

```
print("4110E233 - 高瑞夫")  
  
fruits = ["avocado", "mango", "strawberry"]  
x, y, z = fruits  
  
print(x)  
print(y)  
print(z)
```

```
4110E233 - 高瑞夫  
avocado  
mango  
strawberry
```

# Output Variables

The Python `print()` function is often used to output variables.

```
print("4110E233 - 高瑞夫")  
  
x = "I Love Python"  
print(x)
```

```
4110E233 - 高瑞夫  
I Love Python
```

In the `print()` function, you output multiple variables, separated by a comma:

```
print("4110E233 - 高瑞夫")  
  
x = "I"  
y = "Love"  
z = "Python"  
print(x, y, z)
```

```
4110E233 - 高瑞夫  
I Love Python
```

You can also use the + operator to output multiple variables:

```
print("4110E233 - 高瑞夫")  
  
x = "I "  
y = "Love "  
z = "Python"  
print(x + y + z)
```

```
4110E233 - 高瑞夫  
I Love Python
```

Notice the space character after "I " and "Love ", without them the result would be "ILovePython".

For numbers, the `+` character works as a mathematical operator:

```
print("4110E233 - 高瑞夫")  
  
x = 10  
y = 10  
print(x + y)
```

```
4110E233 - 高瑞夫  
20
```

In the `print()` function, when you try to combine a string and a number with the `+` operator, Python will give you an error:

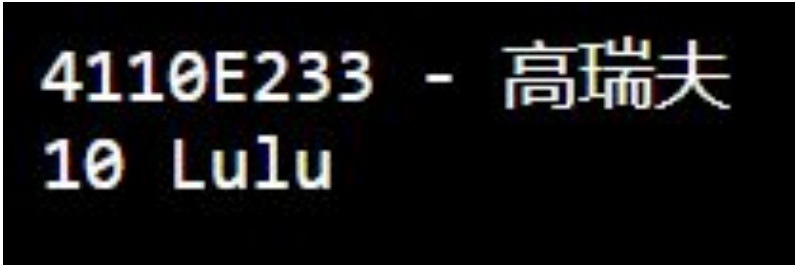
```
print("4110E233 - 高瑞夫")  
  
x = 10  
y = "Lulu"  
print(x + y)
```

```
Traceback (most recent call last):  
  File "./prog.py", line 5, in <module>  
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The best way to output multiple variables in the `print()` function is to separate them with commas, which even support different data types:

```
print("4110E233 - 高瑞夫")
```

```
x = 10  
y = "Lulu"  
print(x, y)
```

A black rectangular box representing a terminal window. It contains two lines of text in a monospaced font. The first line is "4110E233 - 高瑞夫" and the second line is "10 Lulu".

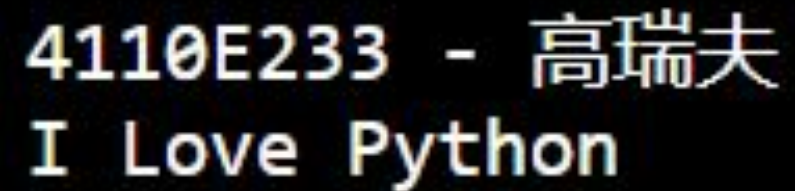
```
4110E233 - 高瑞夫  
10 Lulu
```

# Global Variables

Variables that are created outside of a function (as in all of the examples above) are known as global variables.

Global variables can be used by everyone, both inside of functions and outside.

```
print("4110E233 - 高瑞夫")  
  
x = "Python"  
  
def myfunc():  
    print("I Love " + x)  
  
myfunc()
```



A terminal window showing the output of the Python code. The first line is "4110E233 - 高瑞夫" and the second line is "I Love Python".

```
4110E233 - 高瑞夫  
I Love Python
```



If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function. The global variable with the same name will remain as it was, global and with the original value.

```
print("4110E233 - 高瑞夫")  
  
x = "amazing"  
  
def myfunc():  
    x = "Python"  
    print("I Love " + x)  
  
myfunc()  
  
print("Python is " + x)
```

```
4110E233 - 高瑞夫  
I Love Python  
Python is amazing
```

## ● The global Keyword

Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function.

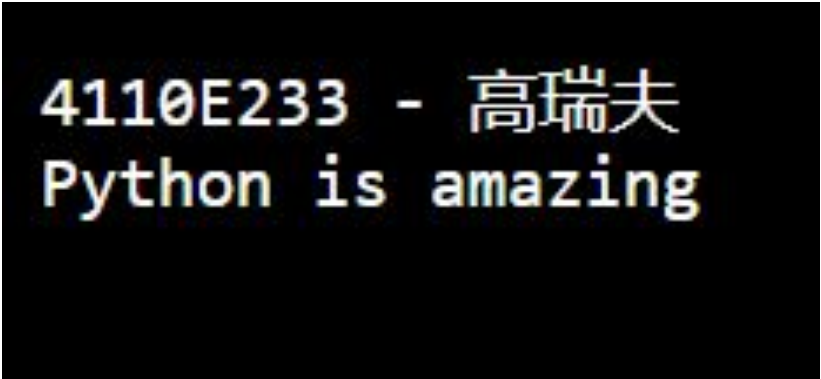
To create a global variable inside a function, you can use the `global` keyword.

```
print("4110E233 - 高瑞夫")

def myfunc():
    global x
    x = "amazing"

myfunc()

print("Python is " + x)
```



```
4110E233 - 高瑞夫
Python is amazing
```

Also, use the **global** keyword if you want to change a global variable inside a function.

```
print("4110E233 - 高瑞夫")

x = "cool"

def myfunc():
    global x
    x = "amazing"

myfunc()

print("Python is " + x)
```

4110E233 - 高瑞夫  
Python is amazing

# PYTHON DATA TYPES

# CONTENT

- ▶ Python Data Types
- ▶ Python Numbers
- ▶ Python Casting
- ▶ Python Strings
  - ▶ Slicing Strings
  - ▶ Modify Strings
  - ▶ Concatenate Strings
  - ▶ Format Strings
  - ▶ Escape Strings
  - ▶ String Methods
- ▶ Python Booleans

# PYTHON DATA TYPES

Data Types

# Data Types

## ● Built-in Data Types

In programming, data type is an important concept.

Variables can store data of different types, and different types can do different things.

Python has the following data types built-in by default, in these categories:

Text Type:	<code>str</code>
Numeric Types:	<code>int, float, complex</code>
Sequence Types:	<code>list, tuple, range</code>
Mapping Type:	<code>dict</code>
Set Types:	<code>set, frozenset</code>
Boolean Type:	<code>bool</code>
Binary Types:	<code>bytes, bytearray, memoryview</code>
None Type:	<code>NoneType</code>

## ● Getting the Data Types

You can get the data type of any object by using the `type()` function:

```
print("4110E233 - 高瑞夫")  
x = 5  
y = True  
z = "Reyb"  
print(type(x))  
print(type(y))  
print(type(z))
```

```
4110E233 - 高瑞夫  
<class 'int'>  
<class 'bool'>  
<class 'str'>
```



## ● Setting the Data Types

In Python, the data type is set when you assign a value to a variable:

### Example

### Data Type

```
x = "Hello World"
```

str

```
x = 20
```

int

```
x = 20.5
```

float

```
x = 1j
```

complex

```
x = ["apple", "banana", "cherry"]
```

list

```
x = ("apple", "banana", "cherry")
```

tuple

```
x = range(6)
```

range

## ● Setting the Data Types

In Python, the data type is set when you assign a value to a variable:

Example	Data Type
<code>x = {"name" : "John", "age" : 36}</code>	dict
<code>x = {"apple", "banana", "cherry"}</code>	set
<code>x = frozenset({"apple", "banana", "cherry"})</code>	frozenset
<code>x = True</code>	bool
<code>x = b"Hello"</code>	bytes
<code>x = bytearray(5)</code>	bytearray
<code>x = memoryview(bytes(5))</code>	memoryview
<code>x = None</code>	NoneType

# PYTHON DATA TYPES

Python Numbers

# Python Numbers

There are three numeric types in Python:

- ▶ int
- ▶ float
- ▶ complex

Variables of numeric types are created when you assign a value to them. To verify the type of any object in Python, use the `type()` function:

```
print("4110E233 - 高瑞夫")  
  
x = 10 # int  
y = 9.9 # float  
z = 10j # complex  
  
print(type(x))  
print(type(y))  
print(type(z))
```

```
4110E233 - 高瑞夫  
<class 'int'>  
<class 'float'>  
<class 'complex'>
```

## ● Int

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

```
print("4110E233 - 高瑞夫")  
  
x = 10  
y = 1234567890  
z = -987654321  
  
print(type(x))  
print(type(y))  
print(type(z))
```

```
4110E233 - 高瑞夫  
<class 'int'>  
<class 'int'>  
<class 'int'>
```

## ● Float

Float, or "floating point number" is a number, positive or negative, containing one or more decimals.

```
print("4110E233 - 高瑞夫")  
  
x = 2.22  
y = 2.0  
z = -22.22  
  
print(type(x))  
print(type(y))  
print(type(z))
```

```
4110E233 - 高瑞夫  
<class 'float'>  
<class 'float'>  
<class 'float'>
```

Float can also be scientific numbers with an "e" to indicate the power of 10.

```
print("4110E233 - 高瑞夫")  
  
x = 99e9  
y = 99E9  
z = -99.9e999  
  
print(type(x))  
print(type(y))  
print(type(z))
```

```
4110E233 - 高瑞夫  
<class 'float'>  
<class 'float'>  
<class 'float'>
```

## ● Complex

Complex numbers are written with a "j" as the imaginary part:

```
print("4110E233 - 高瑞夫")  
  
x = 2+3j  
y = 1j  
z = -1j  
  
print(type(x))  
print(type(y))  
print(type(z))
```

```
4110E233 - 高瑞夫  
<class 'complex'>  
<class 'complex'>  
<class 'complex'>
```



## ● Type Conversion

You can convert from one type to another with the `int()`, `float()`, and `complex()` methods:

```
print("4110E233 - 高瑞夫")

#convert from int to float:
x = float(99)

#convert from float to int:
y = int(123.456)

#convert from int to complex:
z = complex(2)

print(x)
print(y)
print(z)

print(type(x))
print(type(y))
print(type(z))
```

```
4110E233 - 高瑞夫
99.0
123
(2+0j)
<class 'float'>
<class 'int'>
<class 'complex'>
```

**Note:** You cannot convert complex numbers into another number type.

## ● Random Numbers

Python does not have a `random()` function to make a random number, but Python has a built-in module called `random` that can be used to make random numbers:

```
print("4110E233 - 高瑞夫")  
  
import random  
  
print(random.randrange(1, 100))
```

```
4110E233 - 高瑞夫  
97
```

# PYTHON DATA TYPES

Python Casting

# Python Casting

- Specify a Variable Type

There may be times when you want to specify a type on to a variable. This can be done with casting. Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types.

Casting in python is therefore done using constructor functions:

- ▶ `int()` - constructs an integer number from an integer literal, a float literal (by removing all decimals), or a string literal (providing the string represents a whole number)
- ▶ `float()` - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
- ▶ `str()` - constructs a string from a wide variety of data types, including strings, integer literals and float literals

## Example: Integers

```
print("4110E233 - 高瑞夫")  
  
x = int(4)  
y = int(5.6789)  
z = int("6")  
print(x)  
print(y)  
print(z)
```

```
4110E233 - 高瑞夫  
4  
5  
6
```

## Example: Floats

```
print("4110E233 - 高瑞夫")  
  
x = float(9)  
y = float(8.4122141)  
z = float("7")  
w = float("6.123456")  
print(x)  
print(y)  
print(z)  
print(w)
```

```
4110E233 - 高瑞夫  
9.0  
8.4122141  
7.0  
6.123456
```

## Example: Strings

```
print("4110E233 - 高瑞夫")  
  
x = str("s123")  
y = str(123)  
z = str(123.456)  
print(x)  
print(y)  
print(z)
```

```
4110E233 - 高瑞夫  
s123  
123  
123.456
```

# PYTHON DATA TYPES

Python Strings



# Python Strings

- Strings

Strings in python are surrounded by either single quotation marks, or double quotation marks.

'hello' is the same as "hello".

You can display a string literal with the `print()` function:

```
#You can use double or single quotes:  
  
print("4110E233 - 高瑞夫")  
print('4110E233 - 高瑞夫')
```

```
4110E233 - 高瑞夫  
4110E233 - 高瑞夫
```

- Assign Strings to a Variable

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

```
print("高瑞夫 - 4110e233")  
a = "hiyow"  
print(a)
```

```
高瑞夫 - 4110e233  
hiyow
```

## ● Multiline Strings

You can assign a multiline string to a variable by using three quotes:

```
print("高瑞夫 - 4110e233")  
a = """Ako'y may tula  
mahabang mahaba.  
Ako'y uupo  
tapos na po."""  
print(a)
```

```
高瑞夫 - 4110e233  
Ako'y may tula  
mahabang mahaba.  
Ako'y uupo  
tapos na po.
```

**Note:** in the result, the line breaks are inserted at the same position as in the code.

## ● Strings are Arrays

Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.

However, Python does not have a character data type, a single character is simply a string with a length of 1.

Square brackets can be used to access elements of the string.

```
print("高瑞夫 - 4110e233")  
a = "Nin hao, 高瑞夫"  
print(a[9])  
  
# python strings are arrays  
# first character has the position 0
```

高瑞夫 - 4110e233  
高

## ● Looping Through a String

Since strings are arrays, we can loop through the characters in a string, with a **for** loop.

```
print("高瑞夫 - 4110e233")  
for x in "Reyb":  
    print(x)
```

```
高瑞夫 - 4110e233  
R  
e  
y  
b
```

- String Length

To get the length of a string, use the `len()` function.

[illegible]

高瑞夫 - 4110e233  
104

## ● Check String

- ▶ To check if a certain phrase or character is present in a string, we can use the keyword `in`.
- ▶ Keyword `in` vs `if`
- ▶ True or False

```
print("高瑞夫 - 4110e233")
txt = "blah blah blahh"
print("blah" in txt)
print("bleh" in txt)
# print("blah" in txt) == > True
# print("bleh" in txt) == > False
```

```
高瑞夫 - 4110e233
True
False
```

Use it in an `if` statement:

```
print("高瑞夫 - 4110e233")
txt = "blah blah blahh"
if "blahh" in txt:
    print("Yes, 'blahh' is present.")
```

```
高瑞夫 - 4110e233
Yes, 'blahh' is present.
```



- Check if NOT

To check if a certain phrase or character is NOT present in a string, we can use the keyword `not in`.

```
print("高瑞夫 - 4110e233")  
txt = "blah blah blahh"  
print("bleh" not in txt)
```

```
高瑞夫 - 4110e233  
True
```

Use it in an `if` statement:

```
print("高瑞夫 - 4110e233")  
txt = "blah blah blahh"  
if "bleh" not in txt:  
    print("No, 'bleh' is NOT present.")
```

```
高瑞夫 - 4110e233  
No, 'bleh' is NOT present.
```



# Slicing Strings

- Slicing

You can return a range of characters by using the slice syntax.

Specify the start index and the end index, separated by a colon, to return a part of the string.

```
print("高瑞夫 - 4110e233")  
b = "Hello, Laoshi"  
print(b[0:5])  
# The first character has index 0.
```

```
高瑞夫 - 4110e233  
Hello
```

**Note:** The first character has index 0.

- Slice From the Start

By leaving out the start index, the range will start at the first character:

```
print("4110E233 - 高瑞夫")  
b = "Hello, Universe!"  
print(b[:5])
```

```
4110E233 - 高瑞夫  
Hello
```

**Note:** Get the characters from the start to position 5 (not included)

- Slice To the End

By leaving out the *end* index, the range will go to the end:

```
print("高瑞夫 - 4110e233")  
b = "Hello, Reyb!"  
print(b[6:])
```

```
高瑞夫 - 4110e233  
Reyb!
```

**Note:** Get the characters from position 2, and all the way to the end

## ● Negative Indexing

Use negative indexes to start the slice from the end of the string:

```
print("高瑞夫 - 4110e233")  
# Get the characters:  
# From: "R" in "Reyb!" (position -5)  
# To, but not included: "!" in "Reyb!" (position -1)  
b = "Hello, Reyb!"  
print(b[-5:-1])  
# -1 not included
```

高瑞夫 - 4110e233  
Reyb

# Modify Strings

- Upper Case

The `upper()` method returns the string in upper case:

```
print("高瑞夫 - 4110e233")  
a = "Hello, Reyb!"  
print(a.upper())
```

```
高瑞夫 - 4110e233  
HELLO, REYB!
```

- Upper Case

The `lower()` method returns the string in lower case:

```
print("高瑞夫 - 4110e233")  
a = "Hello, Reyb!"  
print(a.lower())
```

```
高瑞夫 - 4110e233  
hello, reyb!
```

## ● Remove Whitespace

- ▶ Whitespace is the space before and/or after the actual text, and very often you want to remove this space.
- ▶ The `strip()` method removes any whitespace from the beginning or the end

```
print("高瑞夫 - 4110e233")  
a = "    Hello, Reyb!    "  
print(a.strip()) # returns "Hello, Reyb!"
```

```
高瑞夫 - 4110e233  
Hello, Reyb!
```

## ● Replace String

The `replace()` method replaces a string with another string

```
print("高瑞夫 - 4110e233")  
b = "Hello, Reyb!"  
print(b.replace("o", "ow"))
```

```
高瑞夫 - 4110e233  
Hellow, Reyb!
```



## ● Split String

The `split()` method returns a list where the text between the specified separator becomes the list items.

```
print("高瑞夫 - 4110e233")  
b = " Hello, Reyb! "  
print(b.split(",")) # returns ['Hello', ' Reyb!']
```

```
高瑞夫 - 4110e233  
[' Hello', ' Reyb! ']
```

# String Concatenation

To concatenate, or combine, two strings you can use the + operator.

```
print("高瑞夫 - 4110e233")  
a = "Hello"  
b = ", "  
c = "Reyb"  
d = "!"  
e = a + b + c + d  
print(e)
```

```
高瑞夫 - 4110e233  
Hello, Reyb!
```

To add a space between them, add a " ":

```
print("高瑞夫 - 4110e233")  
a = "Hello, "  
b = "Reyb!"  
c = a + " " + b  
print(c)
```

```
高瑞夫 - 4110e233  
Hello, Reyb!
```

# Format Strings

As we learned in the Python Variables chapter, we cannot combine strings and numbers like this:

```
print("4110E233 - 高瑞夫")  
  
age = 19  
txt = "My name is Reyb, I am " + age  
print(txt)
```

```
Traceback (most recent call last):  
  File "./prog.py", line 4, in <module>  
TypeError: can only concatenate str (not "int") to str
```

But we can combine strings and numbers by using the `format()` method!

The `format()` method takes the passed arguments, formats them, and places them in the string where the placeholders `{}` are:

```
print("高瑞夫 - 4110e233")  
age = 19  
txt = "My name is Reyb, and I am {}"  
print(txt.format(age))
```

```
高瑞夫 - 4110e233  
My name is Reyb, and I am 19
```

The `format()` method takes unlimited number of arguments, and are placed into the respective placeholders:

```
print("高瑞夫 - 4110e233")
quantity = 10
itemno = 62
price = 99.99
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

```
高瑞夫 - 4110e233
I want 10 pieces of item 62 for 99.99 dollars.
```

You can use index numbers `{0}` to be sure the arguments are placed in the correct placeholders:

```
print("高瑞夫 - 4110e233")
quantity = 10
itemno = 62
price = 99.99
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
print(myorder.format(quantity, itemno, price))
```

```
高瑞夫 - 4110e233
I want to pay 99.99 dollars for 10 pieces of item 62.
```

# Escape Characters

To insert characters that are illegal in a string, use an escape character.

An escape character is a backslash \ followed by the character you want to insert.

An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

```
print("4110E233 - 高瑞夫")
```

```
txt = "I am the so-called \"Black Swordsman\" from the Eastblue."
```

#You will get an error if you use double quotes inside a string that are surrounded by double quotes:

```
Traceback (most recent call last):
  File "/usr/lib/python3.8/py_compile.py", line 144, in compile
    code = loader.source_to_code(source_bytes, dfile or file,
      File "<frozen importlib._bootstrap_external>", line 846, in source_to_code
      File "<frozen importlib._bootstrap>", line 219, in _call_with_frames_removed
      File "./prog.py", line 3
        txt = "I am the so-called "Black Swordsman" from the Eastblue."
            ^
SyntaxError: invalid syntax

During handling of the above exception, another exception occurred:
```

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
    File "/usr/lib/python3.8/py_compile.py", line 150, in compile
      raise py_exc
py_compile.PyCompileError:   File "./prog.py", line 3
      txt = "I am the so-called "Black Swordsman" from the Eastblue."
          ^
SyntaxError: invalid syntax
```



To fix this problem, use the escape character `\`:

```
print("高瑞夫 - 4110e233")  
txt = "I am the so-called \"Black Swordsman\" from the Eastblue."  
print(txt)
```

```
高瑞夫 - 4110e233  
I am the so-called "Black Swordsman" from the Eastblue.
```



# String Methods

**Note:** All string methods return new values. They do not change the original string.

Method	Description
<u>capitalize()</u>	Converts the first character to upper case
<u>casefold()</u>	Converts string into lower case
<u>center()</u>	Returns a centered string
<u>count()</u>	Returns the number of times a specified value occurs in a string
<u>encode()</u>	Returns an encoded version of the string
<u>endswith()</u>	Returns true if the string ends with the specified value
<u>expandtabs()</u>	Sets the tab size of the string
<u>find()</u>	Searches the string for a specified value and returns the position of where it was found
<u>format()</u>	Formats specified values in a string
<u>format_map()</u>	Formats specified values in a string
<u>index()</u>	Searches the string for a specified value and returns the position of where it was found
<u>isalnum()</u>	Returns True if all characters in the string are alphanumeric
<u>isalpha()</u>	Returns True if all characters in the string are in the alphabet

# String Methods

<code>isdecimal()</code>	Returns True if all characters in the string are decimals
<code>isdigit()</code>	Returns True if all characters in the string are digits
<code>isidentifier()</code>	Returns True if the string is an identifier
<code>islower()</code>	Returns True if all characters in the string are lower case
<code>isnumeric()</code>	Returns True if all characters in the string are numeric
<code>isprintable()</code>	Returns True if all characters in the string are printable
<code>isspace()</code>	Returns True if all characters in the string are whitespaces
<code>istitle()</code>	Returns True if the string follows the rules of a title
<code>isupper()</code>	Returns True if all characters in the string are upper case
<code>join()</code>	Joins the elements of an iterable to the end of the string
<code>ljust()</code>	Returns a left justified version of the string
<code>lower()</code>	Converts a string into lower case
<code>lstrip()</code>	Returns a left trim version of the string
<code>maketrans()</code>	Returns a translation table to be used in translations
<code>partition()</code>	Returns a tuple where the string is parted into three parts
<code>replace()</code>	Returns a string where a specified value is replaced with a specified value

# String Methods

<code>rfind()</code>	Searches the string for a specified value and returns the last position of where it was found
<code>rindex()</code>	Searches the string for a specified value and returns the last position of where it was found
<code>rjust()</code>	Returns a right justified version of the string
<code>rpartition()</code>	Returns a tuple where the string is parted into three parts
<code>rsplit()</code>	Splits the string at the specified separator, and returns a list
<code>rstrip()</code>	Returns a right trim version of the string
<code>split()</code>	Splits the string at the specified separator, and returns a list
<code>splitlines()</code>	Splits the string at line breaks and returns a list
<code>startswith()</code>	Returns true if the string starts with the specified value
<code>strip()</code>	Returns a trimmed version of the string
<code>swapcase()</code>	Swaps cases, lower case becomes upper case and vice versa
<code>title()</code>	Converts the first character of each word to upper case
<code>translate()</code>	Returns a translated string
<code>upper()</code>	Converts a string into upper case
<code>zfill()</code>	Fills the string with a specified number of 0 values at the beginning

# PYTHON DATA TYPES

Python Booleans

# Python Booleans

Booleans represent one of two values: **True** or **False**.

- **Booleans Values**

In programming you often need to know if an expression is **True** or **False**.

You can evaluate any expression in Python, and get one of two answers, **True** or **False**.

When you compare two values, the expression is evaluated and Python returns the Boolean answer:

```
print("4110E233 - 高瑞夫")  
  
print(99 > 9)  
print(99 == 9)  
print(99 < 9)
```

```
4110E233 - 高瑞夫  
True  
False  
False
```

When you run a condition in an if statement, Python returns **True** or **False**:

```
print("4110E233 - 高瑞夫")  
  
a = 999  
b = 9999  
  
if b > a:  
    print("b is greater than a")  
else:  
    print("b is not greater than a")
```

```
4110E233 - 高瑞夫  
b is greater than a
```



## ● Evaluate Values and Variables

The `bool()` function allows you to evaluate any value, and give you `True` or `False` in return,

Evaluate a string and a number:

```
print("4110E233 - 高瑞夫")  
print(bool("wuzzup"))  
print(bool(30))
```

```
4110E233 - 高瑞夫  
True  
True
```

Evaluate two variables:

```
print("4110E233 - 高瑞夫")  
  
x = "wuzzup"  
y = 30  
  
print(bool(x))  
print(bool(y))
```

```
4110E233 - 高瑞夫  
True  
True
```

## ● Most Values are True

Almost any value is evaluated to **True** if it has some sort of content.

Any string is **True**, except empty strings.

Any number is **True**, except **0**.

Any list, tuple, set, and dictionary are **True**, except empty ones.

```
print("4110E233 - 高瑞夫")  
print(bool("def"))  
print(bool(456))  
print(bool(["pineapple", "strawberry", "avocado"]))
```

```
4110E233 - 高瑞夫  
True  
True  
True
```



## ● Some Values are False

In fact, there are not many values that evaluate to **False**, except empty values, such as `()`, `[]`, `{}`, `""`, the number `0`, and the value `None`. And of course the value **False** evaluates to **False**.

```
print("4110E233 - 高瑞夫")
print(bool(False))
print(bool(None))
print(bool(0))
print(bool(""))
print(bool(()))
print(bool([]))
print(bool({}))
```

```
4110E233 - 高瑞夫
False
False
False
False
False
False
False
```

One more value, or object in this case, evaluates to **False**, and that is if you have an object that is made from a class with a `__len__` function that returns **0** or **False**:

```
print("4110E233 - 高瑞夫")
class myclass():
    def __len__(self):
        return 0

myobj = myclass()
print(bool(myobj))
```

```
4110E233 - 高瑞夫
False
```

## ● Functions can Return a Boolean

You can create functions that returns a Boolean Value:

```
print("4110E233 - 高瑞夫")  
def myFunction() :  
    return True  
  
print(myFunction())
```

```
4110E233 - 高瑞夫  
True
```

You can execute code based on the Boolean answer of a function:

```
print("4110E233 - 高瑞夫")  
  
def myFunction() :  
    return True  
  
if myFunction():  
    print("YES!")  
else:  
    print("NO!")
```

```
4110E233 - 高瑞夫  
YES!
```

Python also has many built-in functions that return a boolean value, like the `isinstance()` function, which can be used to determine if an object is of a certain data type:

```
print("4110E233 - 高瑞夫")  
x = 200  
print(isinstance(x, int))
```

```
4110E233 - 高瑞夫  
True
```

# PYTHON OPERATION

# CONTENT

- ▶ Python Operators
- ▶ Python Lists
  - ▶ Access List Items
  - ▶ Change List Items
  - ▶ Add List Items
  - ▶ Remove List Items
  - ▶ Loop Lists
  - ▶ List Comprehension
  - ▶ Sort Lists
  - ▶ Copy Lists
  - ▶ Join Lists
  - ▶ List Methods
- ▶ Python Tuples
  - ▶ Access Tuples
  - ▶ Update Tuples
  - ▶ Unpack Tuples

# CONTENT

- ▶ Loop Tuples
- ▶ Join Tuples
- ▶ Tuple Methods
- ▶ Python Sets
  - ▶ Access Set Items
  - ▶ Add Set Items
  - ▶ Remove Set Items
  - ▶ Loop Sets
  - ▶ Join Sets
  - ▶ Set Methods
- ▶ Python Dictionaries
  - ▶ Access Items
  - ▶ Change Items
  - ▶ Add Items
  - ▶ Remove Items
  - ▶ Loop Dictionaries
  - ▶ Copy Dictionaries

# CONTENT

- ▶ Nested Dictionaries
- ▶ Dictionary Methods



# PYTHON OPERATORS

Operators

# Python Operators

Operators are used to perform operations on variables and values.

In the example below, we use the + operator to add together two values:

```
print("4110E233 - 高瑞夫")  
print(1000 + 234)
```

```
4110E233 - 高瑞夫  
1234
```

Python divides the operators in the following groups:

- ▶ Arithmetic operators
- ▶ Assignment operators
- ▶ Comparison operators
- ▶ Logical operators
- ▶ Identity operators
- ▶ Membership operators
- ▶ Bitwise operators

## ● Python Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Name	Example
+	Addition	$x + y$
-	Subtraction	$x - y$
*	Multiplication	$x * y$
/	Division	$x / y$
%	Modulus	$x \% y$
**	Exponentiation	$x ** y$
//	Floor division	$x // y$

## ● Python Assignment Operators

Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
=	x  = 3	x = x   3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

## ● Python Comparison Operators

Comparison operators are used to compare two values:

Operator	Name	Example
<code>==</code>	Equal	<code>x == y</code>
<code>!=</code>	Not equal	<code>x != y</code>
<code>&gt;</code>	Greater than	<code>x &gt; y</code>
<code>&lt;</code>	Less than	<code>x &lt; y</code>
<code>&gt;=</code>	Greater than or equal to	<code>x &gt;= y</code>
<code>&lt;=</code>	Less than or equal to	<code>x &lt;= y</code>

## ● Python Logical Operators

Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	<code>x &lt; 5 and x &lt; 10</code>
or	Returns True if one of the statements is true	<code>x &lt; 5 or x &lt; 4</code>
not	Reverse the result, returns False if the result is true	<code>not(x &lt; 5 and x &lt; 10)</code>

## ● Python Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

## ● Python Membership Operators

Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y



## ● Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

# PYTHON OPERATORS

Python Lists

# Python Lists

<https://github.com/reeeyyybb/cs2022/blob/main/python/1012.md>

# PYTHON OPERATORS

Python Tuples

# Tuple

Tuples are used to store multiple items in a single variable.

Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are [List](#), [Set](#), and [Dictionary](#), all with different qualities and usage.

A tuple is a collection which is ordered and unchangeable.

Tuples are written with round brackets.

```
print("4110E233 - 高瑞夫")  
  
thistuple = ("avocado", "mango", "strawberry")  
print(thistuple)
```

```
4110E233 - 高瑞夫  
( 'avocado', 'mango', 'strawberry' )
```

## ● Tuple Items

Tuple items are ordered, unchangeable, and allow duplicate values.

Tuple items are indexed, the first item has index `[0]`, the second item has index `[1]` etc.

## ● Ordered

When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.

## ● Unchangeable

Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.

## ● Unchangeable

Since tuples are indexed, they can have items with the same value:

```
print("4110E233 - 高瑞夫")  
  
thistuple = ("avocado", "mango", "strawberry", "avocado", "strawberry")  
print(thistuple)
```

```
4110E233 - 高瑞夫  
( 'avocado', 'mango', 'strawberry', 'avocado', 'strawberry' )
```

## ● Tuple Length

To determine how many items a tuple has, use the `len()` function:

```
print("4110E233 - 高瑞夫")  
  
thistuple = tuple(("avocado", "mango", "strawberry"))  
print(len(thistuple))
```

```
4110E233 - 高瑞夫  
3
```



## ● Create Tuple With One Item

To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

```
print("4110E233 - 高瑞夫")

thistuple = ("avocado",)
print(type(thistuple))

#NOT a tuple
thistuple = ("avocado")
print(type(thistuple))
```

```
4110E233 - 高瑞夫
<class 'tuple'>
<class 'str'>
```

## ● Tuple Items - Data Types

Tuple items can be of any data type:

String, int and boolean data types:

```
print("4110E233 - 高瑞夫")

tuple1 = ("avocado", "mango", "strawberry")
tuple2 = (1, 2, 3, 4, 5)
tuple3 = (True, False)

print(tuple1)
print(tuple2)
print(tuple3)
```

```
4110E233 - 高瑞夫
('avocado', 'mango', 'strawberry')
(1, 2, 3, 4, 5)
(True, False)
```

## ● Tuple Items - Data Types

A tuple can contain different data types:

A tuple with strings, integers and boolean values:

```
print("4110E233 - 高瑞夫")  
  
tuple1 = ("avocado", 1, True, 123, "round")  
  
print(tuple1)
```

```
4110E233 - 高瑞夫  
( 'avocado', 1, True, 123, 'round' )
```

- type()

From Python's perspective, tuples are defined as objects with the data type 'tuple':

```
<class 'tuple'>
```

```
print("4110E233 - 高瑞夫")  
mytuple = ("avocado", "mango", "strawberry")  
print(type(mytuple))
```

```
4110E233 - 高瑞夫  
<class 'tuple'>
```

## ● The tuple() Constructor

It is also possible to use the `tuple()` constructor to make a tuple.

```
print("4110E233 - 高瑞夫")  
  
thistuple = tuple(("avocado", "mango", "strawberry"))  
print(thistuple)
```

```
4110E233 - 高瑞夫  
( 'avocado', 'mango', 'strawberry' )
```

# ● Python Collection (Arrays)

There are four collection data types in the Python programming language:

- List is a collection which is ordered and changeable. Allows duplicate members.
- Tuple is a collection which is ordered and unchangeable. Allows duplicate members.
- Set is a collection which is unordered, unchangeable\*, and unindexed. No duplicate members.
- Dictionary is a collection which is ordered\*\* and changeable. No duplicate members.

\*Set *items* are unchangeable, but you can remove and/or add items whenever you like.

\*\*As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

# Access Tuple Items

You can access tuple items by referring to the index number, inside square brackets:

```
print("4110E233 - 高瑞夫")  
thistuple = ("avocado", "mango", "strawberry")  
print(thistuple[0])
```

```
4110E233 - 高瑞夫  
avocado
```

**Note:** The first item has index 0.

## ● Negative Indexing

Negative indexing means start from the end.

-1 refers to the last item, -2 refers to the second last item etc.

```
print("4110E233 - 高瑞夫")  
thistuple = ("avocado", "mango", "strawberry")  
print(thistuple[-2])
```

```
4110E233 - 高瑞夫  
mango
```



## ● Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new tuple with the specified items.

```
print("4110E233 - 高瑞夫")
thistuple = ("avocado", "blueberry", "strawberry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:5])

#This will return the items from position 2 to 5.

#Remember that the first item is position 0,
#and note that the item in position 5 is NOT included
```

```
4110E233 - 高瑞夫
('strawberry', 'orange', 'kiwi')
```

**Note:** The search will start at index 2 (included) and end at index 5 (not included).

## ● Range of Indexes

Remember that the first item has index 0.

By leaving out the start value, the range will start at the first item:

```
print("4110E233 - 高瑞夫")  
thistuple = ("avocado", "mango", "strawberry", "orange", "kiwi", "melon", "mango")  
print(thistuple[:4])
```

```
4110E233 - 高瑞夫  
( 'avocado', 'mango', 'strawberry', 'orange' )
```

## ● Range of Indexes

By leaving out the end value, the range will go on to the end of the list:

```
print("4110E233 - 高瑞夫")  
thistuple = ("avocado", "mango", "strawberry", "orange", "kiwi", "melon", "mango")  
print(thistuple[2:])
```

```
4110E233 - 高瑞夫  
( 'strawberry', 'orange', 'kiwi', 'melon', 'mango' )
```

## ● Range of Negative Indexes

Specify negative indexes if you want to start the search from the end of the tuple:

```
print("4110E233 - 高瑞夫")

thistuple = ("avocado", "mango", "strawberry", "orange", "kiwi", "melon", "mango")
print(thistuple[-4:-1])

#Negative indexing means starting from the end of the tuple.

#This example returns the items from index -4 (included) to index -1 (excluded)

#Remember that the last item has the index -1,
```

```
4110E233 - 高瑞夫
('orange', 'kiwi', 'melon')
```

## ● Check if Item Exists

To determine if a specified item is present in a tuple use the `in` keyword:

```
print("4110E233 - 高瑞夫")  
thistuple = ("avocado", "mango", "strawberry")  
if "mango" in thistuple:  
    print("Yes, 'mango' is in the fruits tuple")
```

```
4110E233 - 高瑞夫  
Yes, 'mango' is in the fruits tuple
```

# PYTHON OPERATORS

Python Sets

# PYTHON OPERATORS

Python Dictionaries

# Python Dictionaries

<https://github.com/reeeyyybb/cs2022/blob/main/python/0914.md>



# PYTHON CONTROLS

# CONTENT

- ▶ Python If ... Else

# PYTHON CONTROLS

Python If ... Else

# Python If ... Else

- Python Conditions and If statements

Python supports the usual logical conditions from mathematics:

- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`

These conditions can be used in several ways, most commonly in "if statements" and loops.

An "if statement" is written by using the `if` keyword.

```
print("4110E233 - 高瑞夫")

a = 111
b = 222

if b > a:
    print("b is greater than a")
```

```
4110E233 - 高瑞夫
b is greater than a
```

In this example we use two variables, `a` and `b`, which are used as part of the if statement to test whether `b` is greater than `a`. As `a` is 111, and `b` is 222, we know that 222 is greater than 111, and so we print to screen that "b is greater than a".

## ● Indentation

Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

```
print("4110E233 - 高瑞夫")

a = 111
b = 222

if b > a:
    print("b is greater than a")
```

```
Traceback (most recent call last):
  File "/usr/lib/python3.8/py_compile.py", line 144, in compile
    code = loader.source_to_code(source_bytes, dfile or file,
  File "<frozen importlib._bootstrap_external>", line 846, in source_to_code
  File "<frozen importlib._bootstrap>", line 219, in _call_with_frames_removed
  File "./prog.py", line 7
    print("b is greater than a")
    ^
IndentationError: expected an indented block

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "/usr/lib/python3.8/py_compile.py", line 150, in compile
    raise py_exc
py_compile.PyCompileError: Sorry: IndentationError: expected an indented block (prog.py, line
```

```
print("4110E233 - 高瑞夫")

a = 111
b = 222

if b > a:
    print("b is greater than a")
```

```
4110E233 - 高瑞夫
b is greater than a
```

## ● Elif

The `elif` keyword is python's way of saying "if the previous conditions were not true, then try this condition".

```
print("4110E233 - 高瑞夫")  
  
a = 123  
b = 123  
if b > a:  
    print("b is greater than a")  
elif a == b:  
    print("a and b are equal")
```

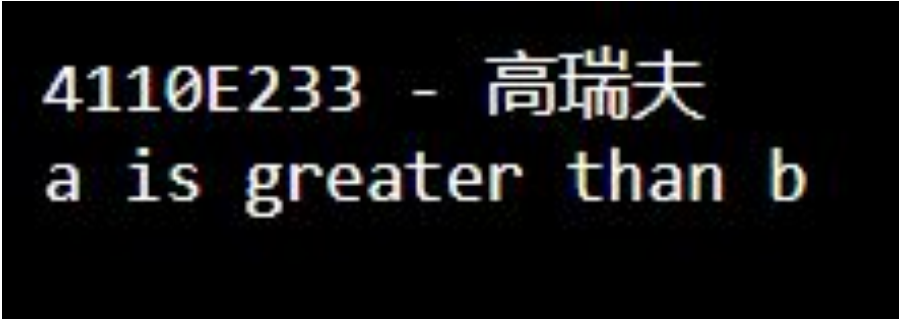
```
4110E233 - 高瑞夫  
a and b are equal
```

In this example `a` is equal to `b`, so the first condition is not true, but the `elif` condition is true, so we print to screen that "a and b are equal".

## ● Else

The `else` keyword catches anything which isn't caught by the preceding conditions.

```
print("4110E233 - 高瑞夫")  
  
a = 123  
b = 12  
if b > a:  
    print("b is greater than a")  
elif a == b:  
    print("a and b are equal")  
else:  
    print("a is greater than b")
```



```
4110E233 - 高瑞夫  
a is greater than b
```

In this example `a` is greater than `b`, so the first condition is not true, also the `elif` condition is not true, so we go to the `else` condition and print to screen that "a is greater than b".



You can also have an `else` without the `elif`:

```
print("4110E233 - 高瑞夫")

a = 123
b = 12
if b > a:
    print("b is greater than a")
else:
    print("b is not greater than a")
```

```
4110E233 - 高瑞夫
b is not greater than a
```

## ● Short Hand If

If you have only one statement to execute, you can put it on the same line as the if statement.

```
print("4110E233 - 高瑞夫")  
  
a = 432  
b = 321  
  
if a > b: print("a is greater than b")
```

```
4110E233 - 高瑞夫  
a is greater than b
```

## ● Short Hand If ... Else

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

```
print("4110E233 - 高瑞夫")  
  
a = 3  
b = 333  
  
print("A") if a > b else print("B")
```

```
4110E233 - 高瑞夫  
B
```

This technique is known as **Ternary Operators**, or **Conditional Expressions**.

You can also have multiple else statements on the same line:

```
print("4110E233 - 高瑞夫")  
  
a = 222  
b = 222  
  
print("A") if a > b else print("=") if a == b else print("B")
```

```
4110E233 - 高瑞夫  
=
```

## ● And

The `and` keyword is a logical operator, and is used to combine conditional statements:

```
print("4110E233 - 高瑞夫")  
  
a = 111  
b = 22  
c = 333  
if a > b and c > a:  
    print("Both conditions are True")
```

```
4110E233 - 高瑞夫  
Both conditions are True
```

## ● Or

The `or` keyword is a logical operator, and is used to combine conditional statements:

```
print("4110E233 - 高瑞夫")  
  
a = 111  
b = 22  
c = 333  
if a > b or a > c:  
    print("At least one of the conditions is True")
```

```
4110E233 - 高瑞夫  
At least one of the conditions is True
```

## ● Nested If

You can have `if` statements inside `if` statements, this is called *nested if* statements.

```
print("4110E233 - 高瑞夫")

x = 30

if x > 12:
    print("Above twelve,")
    if x > 21:
        print("and also above 21!")
    else:
        print("but not above 21.")
```

```
4110E233 - 高瑞夫
Above twelve,
and also above 21!
```

## ● The pass Statement

`if` statements cannot be empty, but if you for some reason have an `if` statement with no content, put in the `pass` statement to avoid getting an error.

```
print("4110E233 - 高瑞夫")  
  
a = 22  
b = 222  
  
if b > a:  
    pass  
  
# having an empty if statement like this, would raise an error without the pass statement
```

4110E233 - 高瑞夫



# PYTHON LOOPS

# CONTENT

- Python While Loops
- Python For Loops

# PYTHON LOOPS

Python While Loops

# Python While Loops

## ● Python Loops

Python has two primitive loop commands:

- `while` loops
- `for` loops

## ● The while Loops

With the `while` loop we can execute a set of statements as long as a condition is true.

```
print("4110E233 - 高瑞夫")

i = 1
while i < 10:
    print(i)
    i += 1
```

```
4110E233 - 高瑞夫
1
2
3
4
5
6
7
8
9
```

**Note:** remember to increment `i`, or else the loop will continue forever.

The `while` loop requires relevant variables to be ready, in this example we need to define an indexing variable, `i`, which we set to 1.

## ● The break Statement

With the `break` statement we can stop the loop even if the while condition is true:

```
print("4110E233 - 高瑞夫")  
  
i = 1  
while i < 10:  
    print(i)  
    if (i == 5):  
        break  
    i += 1
```

```
4110E233 - 高瑞夫  
1  
2  
3  
4  
5
```

## ● The continue Statement

With the `continue` statement we can stop the current iteration, and continue with the next:

```
print("4110E233 - 高瑞夫")

i = 0
while i < 10:
    i += 1
    if i == 5:
        continue
    print(i)

# Note that number 5 is missing in the result
```

4110E233 - 高瑞夫

1  
2  
3  
4  
6  
7  
8  
9  
10

## ● The else Statement

With the `else` statement we can run a block of code once when the condition no longer is true:

```
print("4110E233 - 高瑞夫")

i = 1
while i < 10:
    print(i)
    i += 1
else:
    print("i is no longer less than 10")
```

```
4110E233 - 高瑞夫
1
2
3
4
5
6
7
8
9
i is no longer less than 10
```



# PYTHON LOOPS

Python For Loops

# Python For Loops

A `for` loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the `for` keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the `for` loop we can execute a set of statements, once for each item in a list, tuple, set etc.

```
print("4110E233 - 高瑞夫")  
  
fruits = ["mango", "avocado", "strawberry"]  
for x in fruits:  
    print(x)
```

```
4110E233 - 高瑞夫  
mango  
avocado  
strawberry
```

The `for` loop does not require an indexing variable to set beforehand.

## ● Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

```
print("4110E233 - 高瑞夫")  
  
for x in "chicken":  
    print(x)
```

```
4110E233 - 高瑞夫  
c  
h  
i  
c  
k  
e  
n
```

## ● The break Statement

With the `break` statement we can stop the loop before it has looped through all the items:

```
print("4110E233 - 高瑞夫")  
  
fruits = ["avocado", "mango", "strawberry"]  
for x in fruits:  
    print(x)  
    if x == "mango":  
        break
```

```
4110E233 - 高瑞夫  
avocado  
mango
```

## ● The break Statement

With the `break` statement we can stop the loop before it has looped through all the items:

Exit the loop when `x` is "mango":

```
print("4110E233 - 高瑞夫")  
  
fruits = ["avocado", "mango", "strawberry"]  
for x in fruits:  
    print(x)  
    if x == "mango":  
        break
```

```
4110E233 - 高瑞夫  
avocado  
mango
```

## ● The break Statement

Exit the loop when `x` is "mango", but this time the break comes before the print:

```
print("4110E233 - 高瑞夫")

fruits = ["avocado", "mango", "strawberry"]
for x in fruits:
    if x == "mango":
        break
    print(x)
```

```
4110E233 - 高瑞夫
avocado
```

## ● The continue Statement

With the `continue` statement we can stop the current iteration of the loop, and continue with the next:

```
print("4110E233 - 高瑞夫")  
  
fruits = ["avocado", "mango", "strawberry"]  
for x in fruits:  
    if x == "mango":  
        continue  
    print(x)
```

```
4110E233 - 高瑞夫  
avocado  
strawberry
```

## ● The range() Function

To loop through a set of code a specified number of times, we can use the `range()` function,

The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

```
print("4110E233 - 高瑞夫")  
  
for x in range(10):  
    print(x)
```

```
4110E233 - 高瑞夫  
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

Note that `range(10)` is not the values of 0 to 10, but the values 0 to 9.



## ● The range() Function

The `range()` function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: `range(5, 10)`, which means values from 5 to 10 (but not including 10):

```
print("4110E233 - 高瑞夫")  
  
for x in range(5, 10):  
    print(x)
```

```
4110E233 - 高瑞夫  
5  
6  
7  
8  
9
```

## ● The range() Function

The `range()` function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: `range(4, 44, 4)`:

```
print("4110E233 - 高瑞夫")  
  
for x in range(4, 44, 4):  
    print(x)
```

```
4110E233 - 高瑞夫  
4  
8  
12  
16  
20  
24  
28  
32  
36  
40
```

## ● Else in For Loop

The `else` keyword in a `for` loop specifies a block of code to be executed when the loop is finished:

Print all numbers from 0 to 9, and print a message when the loop has ended:

```
print("4110E233 - 高瑞夫")  
  
for x in range(10):  
    print(x)  
else:  
    print("Finished!")
```

```
4110E233 - 高瑞夫  
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
Finished!
```

## ● Else in For Loop

**Note:** The `else` block will NOT be executed if the loop is stopped by a `break` statement.

Break the loop when `x` is 5, and see what happens with the `else` block:

```
print("4110E233 - 高瑞夫")  
  
for x in range(10):  
    if x == 5: break  
    print(x)  
else:  
    print("Finished!")  
  
#If the loop breaks, the else block is not executed.
```

```
4110E233 - 高瑞夫  
0  
1  
2  
3  
4
```

## ● Nested Loops

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":

```
print("4110E233 - 高瑞夫")  
  
adj = ["black", "huge", "yummy"]  
fruits = ["avocado", "mango", "strawberry"]  
  
for x in adj:  
    for y in fruits:  
        print(x, y)
```

```
4110E233 - 高瑞夫  
black avocado  
black mango  
black strawberry  
huge avocado  
huge mango  
huge strawberry  
yummy avocado  
yummy mango  
yummy strawberry
```

## ● The pass Statement

`for` loops cannot be empty, but if you for some reason have a `for` loop with no content, put in the `pass` statement to avoid getting an error.

```
print("4110E233 - 高瑞夫")  
  
for x in [6, 7, 8]:  
    pass  
  
# having an empty for loop like this, would raise an error without the pass statement
```

4110E233 - 高瑞夫

# PYTHON FUNCTIONS

# CONTENT

- Python Functions
- Python Lambda



# PYTHON FUNCTIONS

Functions

# Python Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

## ● Creating a Function

In Python a function is defined using the `def` keyword:

```
def my_function():  
    print("Hello function")
```

## ● Calling a Function

To call a function, use the function name followed by parenthesis:

```
print("4110E233 - 高瑞夫")  
  
def my_function():  
    print("Hello function")  
  
my_function()
```

```
4110E233 - 高瑞夫  
Hello function
```

## ● Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

```
print("4110E233 - 高瑞夫")

def my_function(fname):
    print(fname + " Bills")

my_function("Electric")
my_function("Wotoh")
my_function("Gloss")
```

```
4110E233 - 高瑞夫
Electric Bills
Wotoh Bills
Gloss Bills
```

*Arguments* are often shortened to *args* in Python documentations.

## ● Parameters or Arguments

The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that is sent to the function when it is called.

## ● Parameters or Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

```
print("4110E233 - 高瑞夫")

def my_function(fname, lname):
    print(fname + " " + lname)

my_function("Raul", "Leyman")
```

```
4110E233 - 高瑞夫
Raul Leyman
```

If you try to call the function with 1 or 3 arguments, you will get an error:

```
print("4110E233 - 高瑞夫")

def my_function(fname, lname):
    print(fname + " " + lname)

my_function("Raul")
```

```
Traceback (most recent call last):
  File "./prog.py", line 6, in <module>
TypeError: my_function() missing 1 required positional argument: 'lname'
```

## ● Arbitrary Arguments, \*args

If you do not know how many arguments that will be passed into your function, add a `*` before the parameter name in the function definition.

This way the function will receive a *tuple* of arguments, and can access the items accordingly:

```
print("4110E233 - 高瑞夫")

def my_function(*kids):
    print("The oldest child is " + kids[0])

my_function("Electro", "Wotoh", "Gloss")
```

```
4110E233 - 高瑞夫
The oldest child is Electro
```

*Arbitrary Arguments* are often shortened to *\*args* in Python documentations.

## ● Keyword Arguments

You can also send arguments with the *key = value* syntax.

This way the order of the arguments does not matter.

```
print("4110E233 - 高瑞夫")  
  
def my_function(child3, child2, child1):  
    print("The oldest child is " + child1)  
  
my_function(child1 = "Electro", child2 = "Wotoh", child3 = "Gloss")
```

```
4110E233 - 高瑞夫  
The oldest child is Electro
```

The phrase *Keyword Arguments* are often shortened to *kwargs* in Python documentations.



## ● Arbitrary Keyword Arguments, \*\*kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: **\*\*** before the parameter name in the function definition.

This way the function will receive a *dictionary* of arguments, and can access the items accordingly:

```
print("4110E233 - 高瑞夫")

def my_function(**kid):
    print("His first name is " + kid["fname"])

my_function(fname = "Gloss", lname = "Bills")
```

```
4110E233 - 高瑞夫
His first name is Gloss
```

*Arbitrary Kword Arguments* are often shortened to **\*\*kwargs** in Python documentations.

## ● Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without argument, it uses the default value:

```
print("4110E233 - 高瑞夫")

def my_function(country = "Philippines"):
    print("I am from " + country)

my_function("Taiwan")
my_function("America")
my_function()
my_function("Japan")
```

```
4110E233 - 高瑞夫
I am from Taiwan
I am from America
I am from Philippines
I am from Japan
```

## ● Passing a List as an Argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:

```
print("4110E233 - 高瑞夫")

def my_function(food):
    for x in food:
        print(x)

fruits = ["avocado", "mango", "strawberry"]

my_function(fruits)
```

```
4110E233 - 高瑞夫
avocado
mango
strawberry
```

## ● Return Values

To let a function return a value, use the `return` statement:

```
print("4110E233 - 高瑞夫")  
  
def my_function(x):  
    return 9 * x  
  
print(my_function(10))  
print(my_function(100))  
print(my_function(1000))
```

```
4110E233 - 高瑞夫  
90  
900  
9000
```

## ● The pass Statement

`function` definitions cannot be empty, but if you for some reason have a `function` definition with no content, put in the `pass` statement to avoid getting an error.

```
print("4110E233 - 高瑞夫")  
  
def myfunction():  
    pass  
  
# having an empty function definition like this, would raise an error without the pass statement
```

```
4110E233 - 高瑞夫
```

## ● Recursion

Python also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

In this example, `tri_recursion()` is a function that we have defined to call itself ("recurse"). We use the `k` variable as the data, which decrements (`-1`) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

## ● Recursion

```
print("4110E233 - 高瑞夫")

def tri_recursion(k):
    if(k > 0):
        result = k + tri_recursion(k - 1)
        print(result)
    else:
        result = 0
    return result

print("\n\nRecursion Example Results")
tri_recursion(10)
```

4110E233 - 高瑞夫

Recursion Example Results

1  
3  
6  
10  
15  
21  
28  
36  
45  
55

# PYTHON FUNCTIONS

Python Lambda



# Python Lambda

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

## ● Syntax

```
lambda arguments : expression
```

The expression is executed and the result is returned:

### Example

Add 990 to argument `a`, and return the result:

```
print("4110E233 - 高瑞夫")  
  
x = lambda a: a + 990  
print(x(10))
```

```
4110E233 - 高瑞夫  
1000
```

## ● Syntax

Lambda functions can take any number of arguments:

### Example

Multiply argument `a` with argument `b` and return the result:

```
print("4110E233 - 高瑞夫")  
  
x = lambda a, b: a * b  
print(x(10, 20))
```

```
4110E233 - 高瑞夫  
200
```

## ● Syntax

Lambda functions can take any number of arguments:

### Example

Summarize argument `a`, `b`, and `c` and return the result:

```
print("4110E233 - 高瑞夫")  
  
x = lambda a, b, c: a + b + c  
print(x(3, 3, 3))
```

```
4110E233 - 高瑞夫  
9
```

## ● Why Use Lambda Function

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

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

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

Use that function definition to make a function that always doubles the number you send in:

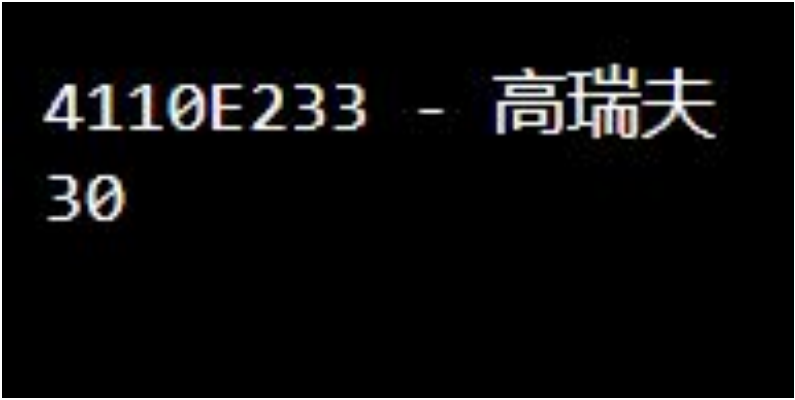
```
print("4110E233 - 高瑞夫")  
  
def myfunc(n):  
    return lambda a : a * n  
  
mydoubler = myfunc(2)  
  
print(mydoubler(10))
```

```
4110E233 - 高瑞夫  
20
```

## ● Why Use Lambda Function

Or, use the same function definition to make a function that always *triples* the number you send in:

```
print("4110E233 - 高瑞夫")  
  
def myfunc(n):  
    return lambda a : a * n  
  
mytripler = myfunc(3)  
  
print(mytripler(10))
```



4110E233 - 高瑞夫  
30

The image shows a terminal window with a black background and white text. The first line of output is "4110E233 - 高瑞夫", which matches the first line of the code. The second line of output is "30", which is the result of the lambda function (3 \* 10) being called.

## ● Why Use Lambda Function

Or, use the same function definition to make both functions, in the same program:

```
print("4110E233 - 高瑞夫")

def myfunc(n):
    return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(10))
print(mytripler(10))
```

```
4110E233 - 高瑞夫
20
30
```

to be continued...