# 3. Getting Started

### First Python Script - `hello.py`

Use a programming text editor to write the following Python script and save as "`hello.py`" in a directory of your choice:

```
print('Hello, world')          # Print a string
print(2 ** 88)                 # Print 2 raises to the power of 88
                               # Python's integer is unlimited in size!
print(8.01234567890123456789) # Print a float
print((1+2j) * (3*4j))         # Python supports complex numbers!
```

Program Notes:

- Statements beginning with a `#` until the end-of-line are comments.
- The `print()` function can be used to print a value to the console.
- Python's strings can be enclosed with single quotes `'...'` (Line 1) or double quotes `"..."`.
- Python's integer is unlimited in size (Line 2).
- Python support floats (Line 4).
- Python supports complex numbers (Line 5) and other high-level data types.
- By convention, Python script (module) filenames are in all-lowercase.

The expected outputs in Python 2 are:

```
Hello, world
309485009821345068724781056
8.0123456789
(-24+12j)
```

The expected outputs in Python 3 are:

```
Hello, world
309485009821345068724781056
8.012345678901234
(-24+12j)
```

### Running Python Scripts

You can develop/run a Python script in many ways - explained in the following sections.

### Running Python Scripts via System Command Shell

You can run a python script via the Python Interpreter under the System's Command Shell (e.g., Windows Command Prompt, Linux/UNIX/Mac OS Terminal/Bash Shell).

- In Linux/Mac OS Bash Shell:

```
$ cd <dirname>     # Change directory to where you stored the script
$ python hello.py  # Run the script via the Python 2 interpreter (or "python3 hello.py")
```

- In Windows Command Prompt: Start a CMD by entering "`cmd`" in the start menu.

```
> cd <dirname>     # Change directory to where you stored the script
> python hello.py  # Run the script via the Python Interpreter
> hello.py         # if ".py" file is associated with Python Interpreter
```

### Unix Executable Shell Script

In Linux/UNIX/Mac OS, you can turn a Python script into an executable program (called Shell Script or Executable Script) by:

1. Start with a line beginning with `#!` (called "hash-bang" or "she-bang"), followed by the full-path name to the Python Interpreter, e.g.,

```
#!/usr/bin/python3
print('Hello, world')
print(2 ** 88)
print(8.01234567890123456789)
print((1+2j) * (3*4j))
```

   To locate the Python Interpreter, use command "`which python`" or "`which python3`".

2. Make the file executable via `chmod` (change file mode) command:

```
$ cd /path/to/project-directory
$ chmod u+x hello.py  # enable executable for user-owner
$ ls -l hello.py      # list to check the executable flag
-rwxrw-r-- 1 uuuu gggg 314 Nov  4 13:21 hello.py
```

3. You can then run the Python script just like any executable program. The system will look for the Python Interpreter from the she-bang line.

```
$ cd /path/to/project-directory
$ ./hello.py
```

The drawback is that you have to hard code the path to the Python Interpreter, which may prevent the program from being portable across different machines.

# 4. Python IDEs and Debuggers

Picking a IDE with a powerful debugger is CRITICAL in program development!!!

## *4.1 IDLE*

Python IDLE (Interactive DeveLopment Environment) is a simple IDE with features such as syntax highlighting, automatic code indentation and debugger. I strongly recommend it for *learning* Python.

### Installing/Launching IDLE

- For Ubuntu: To install Python IDLE for Python 2 and Python 3, respectively:

```
# Install IDLE for Python 2
$ sudo apt-get install idle
# Install IDLE for Python 3
$ sudo apt-get install idle3
```

To launch IDLE for Python 2 or Python 3:

```
$ idle
$ idle3
```

- For Windows: IDLE is bundled in the installation. Click the START button ⇒ Python ⇒ IDLE (Python GUI). To exit, choose "File" menu ⇒ Exit. IDLE is written in Python and is kept under "`Lib\idlelib`". You can also use "`idle.bat`", "`idle.py`", "`idle.pyw`" to start the IDLE.

### Writing Python Script in IDLE

To write a Python script under IDLE, choose "File" menu ⇒ "New File". Enter the above script and save as "`hello.py`" in a directory of your choice (the Python script must be saved with the "`.py`" extension). To run the script, choose **"Run" Menu ⇒ "Run Module".** You shall see the outputs on the IDLE console.

Notes:

- You can use **Alt-P/Alt-N** to retrieve the previous/next command in the command history.
- You can read the Python Manual via the "Help" menu.

### Debugging Python Script in IDLE

In the main window of IDLE console, choose **"Debug" ⇒ "Debugger"** to pop out the "Debug Control". In the text edit window, choose **"Run" ⇒ "Run Module"** to start debugging the script. You can then **step over (Over), step into (Step), step out (Out)** through the script from the "Debug Control". You can also set breakpoint by right clicking on the editor window.

# 5. Python Basics

## *5.1 Python Syntax*

### Comments

A Python comment begins with a hash sign (#) and last till the end of the current line. Comments are ignored by the Python Interpreter, but they are critical in providing explanation and documentation for others (and yourself) to read your program.

Multi-line comment begins with ( ' ' ' ) and last till the other ( ' ' ' ). Also (""") and (""") are used for multiline comments.

```
#Single line comment

'''
multi-line
comment
'''

"""
also,
multi-line comment
"""
```

### Statements

A Python statement is delimited by a newline. A statement cannot cross line boundaries, except:

1. An expression in parentheses (), square bracket [], and curly braces {} can span multiple lines.
2. A backslash (\) at the end of the line denotes continuation to the next line. This is an old rule and is NOT recommended as it is error-prone.

Unlike C/C++/Java, you don't place a semicolon (;) at the end of a Python statement. But you can place multiple statements on a single line, separated by semicolon (;). For examples,

```
# One Python statement in one line.
# A Python statement is terminated by a newline.
# There is no semicolon at the end of a statement.
>>> x = 1     # Assign variable x to 1
>>> print(x)  # Print the value of x
1

# You can place multiple statements in one line, separated by semicolon
>>> print(x); print(x+1); print(x+2)
1
2
3

# An expression in brackets can span multiple lines
>>> x = [1,
         22,
```

```
         333]  # Re-assign variable x to a list
>>> print(x)
[1, 22, 333]

# To break a long expression into several lines, enclosed it with parentheses
>>> x =(1 +
        2
        + 3)
>>> x
6

# You can break a long string into several lines with parentheses too
>>> s = ('testing '   # No commas needed
         'hello, '
         'world!')
>>> s
'testing hello, world!'
```

### Block and Indentation

A block is a group of statements executing as a unit. **Unlike C/C++/Java, which use braces {} to group statements in a body block, Python uses indentation for body block.** In other words, indentation is syntactically significant in Python - the body block must be properly indented. This is a good syntax to force you to indent the blocks correctly for ease of understanding!!!

### Compound Statements

A compound statement, such as `def` (function definition) and `while` loop, begins with a header line terminated with a colon (`:`); followed by the indented body block. Python does not specify how much indentation to use, but all statements of the SAME body block must start at the SAME distance from the right margin. You can use either space or tab for indentation but you cannot mix them in the SAME body block. It is recommended to use 3 spaces (or 4 spaces) for each indentation level. For example,

```
# Define the function main()
def main():
    """Main function"""
    print(sum_1_to_n(100))

# Define the function sum_1_to_n()
def sum_1_to_n(n):
    """Sum from 1 to the given n"""
    sum = 0;
    i = 0;
```

```
    while (i <= n):
        sum += i
        i += 1
    return sum

# Invoke function main()
main()
```

Notes:

- Use IDLE to create the above script called "`SumNumber.py`" (File ⇒ New File). Run the script (Run ⇒ Run Module).
- We define two functions: `main()` and `sum_1_to_n()`, via two `def` compound statements.
- The trailing colon (`:`) signals the start of a body block. All statements belonging to the SAME block must be indented at the SAME distance from the right margin.
- The first line of the function body block is a documentation string (or *doc-string*), followed by the function definition statements.

The trailing colon (`:`) and body indentation is probably the most strange feature in Python, if you come from C/C++/Java. Python imposes strict indentation rules to force programmers to write readable codes!

## *5.2 Naming Conventions and Coding Styles*

These are the recommended naming conventions in Python:

- Variable names: use a noun in lowercase words (optionally joined with underscore *if it improves readability*), e.g., `num_students`.
- Function names: use a verb in lowercase words (optionally joined with underscore *if it improves readability*), e.g., `getarea()` or `get_area()`.
- Class names: use a noun in camel-case (initial-cap all words), e.g., `MyClass`, `IndexError`.
- Constant names: use a noun in uppercase words joined with underscore, e.g., `PI`, `MAX_STUDENTS`.

### Coding Styles

The recommended styles are:

- Use 4 spaces for indentation. Don't use tab.
- Lines shall not exceed 79 characters.
- Use blank lines to separate functions and classes.
- Use a space before and after an operator.

## 5.3 Console Input/Output: Functions `input()` and `print()`

You can use function `input()` to read input from the console (as a string) and `print()` to print output to the console. For example,

```
>>> x = input('Enter a number: ')
Enter a number: 5
>>> x
'5'
>>> type(x)
<class 'str'>
>>> print(x)
5

# Test print()
>>> print('apple')
apple
>>> print('apple', 'orange')  # More items separated by commas
apple orange
>>> print('apple', 'orange', 'banana')
apple orange banana
```

### `print()` without newline

The `print()` function prints a newline at the end of the output by default. You can use the keyword argument "`end`" to specify another delimiter (Python 3). For examples,

```
>>> for i in range(5):
        print(i)  # default a newline at the end
0
1
2
3
4
>>> for i in range(5):
        print(i, end=',')  # print a comma at the end
0,1,2,3,4,
>>> for i in range(5):
        print(i, end='--')
0--1--2--3--4--
```

```
>>> for i in range(5):
       print(i, end='')   # nothing at the end
01234
```

### `print` in Python 2 vs Python 3

Recall that Python 2 and Python 3 are NOT compatible. In Python 2, you can use "`print item`", without the parentheses (because `print` is a keyword in Python 2). In Python 3, parentheses are required as `print()` is a function. For example,

```
# Python 3
>>> print('hello')
hello
>>> print 'hello'
  File "<stdin>", line 1
    print 'hello'
                ^
SyntaxError: Missing parentheses in call to 'print'
>>> print('aaa', 'bbb')
aaa bbb
   # Treated as multiple arguments, printed without parentheses

# Python 2
>>> print('Hello')
Hello
>>> print 'hello'
hello
>>> print('aaa', 'bbb')
('aaa', 'bbb')
   # Treated as a tuple (of items). Print the tuple with parentheses
>>> print 'aaa', 'bbb'
aaa bbb
   # Treated as multiple arguments
```

**IMPORTANT**: Always use `print()` function with parentheses, for portability!

## *5.4 Source Code Encoding*

To specify the character encoding scheme of your source code, e.g., in UTF-8:

```
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-
```

```
......
```

The default encoding is 7-bit ASCII.

I strongly encourage you to encode in UTF-8 for internationalization (i18n).

# 6. Data Types and Dynamic Typing

Python has a large number of built-in data types, such as Numbers (Integer, Float, Boolean, Complex Number), String, List, Tuple, Set, Dictionary and File. More high-level data types, such as Decimal and Fraction, are supported by external modules.

## *6.1 Number Types*

Python supports these built-in number types:

1. Integers (type `int`): e.g., `123`, `-456`. **Unlike C/C++/Java, integers are of unlimited size in Python**. For example,

```
>>> 123 + 456 - 789
-210
>>> 12345678901234567890123456789012345678901 + 1
12345678901234567890123456789012345678902
>>> 123456789012345678901234567890123456789012345678901 + 1
123456789012345678901234567890123456789012345678902
>>> 2 ** 888     # Raise 2 to the power of 888
......
>>> len(str(2 ** 888))  # Convert integer to string and get its length
268                     # 2 to the power of 888 has 268 digits
>>> type(123)    # Get the type
<class 'int'>
>>> help(int)    # Show the help menu for type int
```

   You can also express integers in hexadecimal with prefix `0x` (or `0X`); in octal with prefix `0o` (or `0O`); and in binary with prefix `0b` (or `0B`). For examples, `0x1abc`, `0X1ABC`, `0o1776`, `0b11000011`.

2. Floating-point numbers (type `float`): e.g., `1.0`, `-2.3`, `3e4`, `-3E-4`, with a decimal point and an optional exponent (in `e` or `E`). Floats are 64-bit double precision floating-point numbers. For example,

```
>>> 1.23 * -4e5
-492000.0
>>> type(1.2)        # Get the type
<class 'float'>
```

```
>>> import math      # Using the math module
>>> math.pi
3.141592653589793
>>> import random    # Using the random module
>>> random.random()  # Generate a random number in [0, 1)
0.890839384187198
```

3. Booleans (type `bool`): takes a value of either `True` or `False` (take note of the spelling in initial-capitalized).

```
>>> 8 == 8      # Compare
True
>>> 8 == 9
False
>>> type(True)  # Get type
<class 'bool'>
```

In Python, integer `0`, an empty value (such as empty string `''`, `""`, empty list `[]`, empty tuple `()`, empty dictionary `{}`), and `None` are treated as `False`; anything else are treated as `True`. Booleans can also act as integers in arithmetic operations with `1` for `True` and `0` for `False`. For example,

```
>>> bool(0)
False
>>> bool(1)
True
>>> True + 3
4
>>> False + 1
1
```

4. Complex Numbers (type `complex`): e.g., `1+2j`, `-3-4j`. Complex numbers have a real part and an imaginary part denoted with suffix of `j` (or `J`). For example,

```
>>> x = 1 + 2j  # Assign variable x to a complex number
>>> x           # Display x
(1+2j)
>>> x.real      # Get the real part
1.0
>>> x.imag      # Get the imaginary part
2.0
>>> type(x)     # Get type
<class 'complex'>
>>> x * (3 + 4j)  # Multiply two complex numbers
```

```
(-5+10j)
```

5. Other number types are provided by external modules, such as `decimal` module for decimal fixed-point numbers, `fraction` module for rational numbers.

```
# floats are imprecise
>>> 0.1 * 3
0.30000000000000004

# Decimal are precise
>>> import decimal  # Using the decimal module
>>> x = decimal.Decimal('0.1')  # Construct a Decimal object
>>> x * 3    # Multiply  with overloaded * operator
Decimal('0.3')
>>> type(x)  # Get type
<class 'decimal.Decimal'>
```

## 6.2 The None Value

Python provides a special value called None (take note of the spelling in initial-capitalized), which can be used to initialize an object (to be discussed in OOP later). For example,

```
>>> x = None
>>> type(x)   # Get type
<class 'NoneType'>
>>> print(x)
None

# Use 'is' and 'is not' to check for 'None' value.
>>> print(x is None)
True
>>> print(x is not None)
False
```

## 6.3 Dynamic Typing and Assignment Operator

Like most of the scripting languages (such as Perl, JavaScript, PHP) and unlike general-purpose programming language (such as C/C++/Java/C#), Python is *dynamic typed*. It associates types with objects, instead of variables. That is, a

variable does not have a fixed type and can be assigned an object of any type. A variable simply provides a *reference* to an object.

You do not need to *declare* a variable. A variable is created automatically when a value is first assigned, which links the object to the variable. You can use built-in function `type(var_name)` to get the object type referenced by a variable.

```
>>> x = 1          # Assign an int value to create variable x
>>> x              # Display x
1
>>> type(x)        # Get the type of x
<class 'int'>
>>> x = 1.0        # Re-assign x to a float
>>> x
1.0
>>> type(x)        # Show the type
<class 'float'>
>>> x = 'hello'    # Re-assign x to a string
>>> x
'hello'
>>> type(x)        # Show the type
<class 'str'>
>>> x = '123'      # Re-assign x to a string (of digits)
>>> x
'123'
>>> type(x)        # Show the type
<class 'str'>
```

### Type Conversion

You can perform type conversion via built-in functions `int()`, `float()`, `str()`, `bool()`, etc. For example,

```
>>> x = '123'
>>> type(x)
<class 'str'>
>>> x = int(x)     # Parse str to int, and assign back to x
>>> x
123
>>> type(x)
<class 'int'>
>>> x = float(x)   # Convert x from int to float, and assign back to x
>>> x
123.0
```

```
>>> type(x)
<class 'float'>
>>> x = str(x)    # Convert x from float to str, and assign back to x
>>> x
'123.0'
>>> type(x)
<class 'str'>
>>> len(x)        # Get the length of the string
5
>>> x = bool(x)   # Convert x from str to boolean, and assign back to x
>>> x             # Non-empty string is converted to True
True
>>> type(x)
<class 'bool'>
>>> x = str(x)    # Convert x from bool to str
>>> x
'True'
```

In summary, a variable does not associate with a type. Instead, a type is associated with an object. A variable provides a reference to an object (of a certain type).

### The Assignment Operator (=)

In Python, you do not need to *declare* variables before they are used. The initial assignment creates a variable and links the value to the variable. For example,

```
>>> x = 8        # Create a variable x by assigning a value
>>> x = 'Hello'  # Re-assign a value (of a different type) to x

>>> y            # Cannot access undefined (unassigned) variable
NameError: name 'y' is not defined
```

### `del`

You can use `del` statement to delete a variable. For example,

```
>>> x = 8     # Create variable x via assignment
>>> x
8
>>> del x     # Delete variable x
```

```
>>> x
NameError: name 'x' is not defined
```

### Pair-wise Assignment and Chain Assignment

For example,

```
>>> a = 1  # Ordinary assignment
>>> a
1
>>> b, c, d = 123, 4.5, 'Hello'  # Pair-wise assignment of 3 variables and values
>>> b
123
>>> c
4.5
>>> d
'Hello'
>>> e = f = g = 123  # Chain assignment
>>> e
123
>>> f
123
>>> g
123
```

Assignment operator is *right-associative*, i.e., `a = b = 123` is interpreted as `(a = (b = 123))`.

## 6.4 Number Operations

### Arithmetic Operators

Python supports these arithmetic operators:

| Operator | Description | Examples |
|---|---|---|
| + | Addition | |
| - | Subtraction | |
| * | Multiplication | |
| / | Float Division<br>(returns a float) | 1 / 2 ⇒ 0.5<br>-1 / 2 ⇒ -0.5 |
| // | Integer Division<br>(returns the floor integer) | 1 // 2 ⇒ 0<br>-1 // 2 ⇒ -1<br>8.9 // 2.5 ⇒ 3.0 |

| Operator | Description | Examples |
|---|---|---|
| | | `-8.9 // 2.5 ⇒ -4.0`<br>`-8.9 // -2.5 ⇒ 3.0` |
| ** | Exponentiation | `2 ** 5 ⇒ 32`<br>`1.2 ** 3.4 ⇒ 1.858729691979481` |
| % | Modulus (Remainder) | `9 % 2 ⇒ 1`<br>`-9 % 2 ⇒ 1`<br>`9 % -2 ⇒ -1`<br>`-9 % -2 ⇒ -1`<br>`9.9 % 2.1 ⇒ 1.5`<br>`-9.9 % 2.1 ⇒ 0.6000000000000001` |

Notes:

- **Python does not support increment (++) and decrement (--) operators (as in C/C++/Java).** You need to use `i = i + 1` or `i += 1` for increment.
- Each of the operators has a corresponding *shorthand assignment* counterpart, i.e., +=, -=, *=, /=, //=, **= and %=. For example `i += 1` is the same as `i = i + 1`.
- For mixed-type operations, e.g., 1 + 2.3 (int + float), the value of the "smaller" type is first promoted to the "bigger" type. It then performs the operation in the "bigger" type and returns the result in the "bigger" type. In Python, `int` is "smaller" than `float`, which is "smaller" than `complex`.

### Bitwise Operators

Python supports these bitwise operators:

| Operator | Description | Example<br>`x=0b10000001`<br>`y=0b10001111` |
|---|---|---|
| & | bitwise AND | `x & y ⇒ 0b10000001` |
| \| | bitwise OR | `x \| y ⇒ 0b10001111` |
| ~ | bitwise NOT (or negate) | `~x ⇒ -0b10000010` |
| ^ | bitwise XOR | `x ^ y ⇒ 0b00001110` |
| << | bitwise Left-Shift (padded with zeros) | `x << 2 ⇒ 0b1000000100` |
| >> | bitwise Right-Shift (padded with zeros) | `x >> 2 ⇒ 0b100000` |

### Built-in Functions

Python provides many built-in functions for numbers, including:

- Mathematical functions: `round()`, `pow()`, `abs()`.
- Type conversion functions: `int()`, `float()`, `str()`, `bool()`; and `type()` to get the type.
- Base radix conversion functions: `hex()`, `bin()`, `oct()`.

```
# Test built-in function round()
>>> x = 1.23456
>>> type(x)
<type 'float'>

# Python 3
>>> round(x)     # Round to the nearest integer
1
>>> type(round(x))
<class 'int'>

# Python 2
>>> round(x)
1.0
>>> type(round(x))
<type 'float'>

>>> round(x, 1)  # Round to 1 decimal place
1.2
>>> round(x, 2)  # Round to 2 decimal places
1.23
>>> round(x, 8)  # No change - not for formatting
1.23456

# Test other built-in functions
>>> pow(2, 5)
32
>>> abs(-4.1)
4.1

# Base radix conversion
>>> hex(1234)
'0x4d2'
>>> bin(254)
'0b11111110'
>>> oct(1234)
'0o2322'
>>> 0xABCD  # Shown in decimal by default
43981

# List built-in functions
>>> dir(__builtins__)
['type', 'round', 'abs', 'int', 'float', 'str', 'bool', 'hex', 'bin', 'oct',......]

# Show number of built-in functions
>>> len(dir(__builtins__))  # Python 3
151
>>> len(dir(__builtins__))  # Python 2
144

# Show documentation of __builtins__ module
>>> help(__builtins__)
```

### Relational (Comparison) Operators

Python supports these relational (comparison) operators that return a `bool` value of either `True` or `False`.

| Operator | Description |
|---|---|
| `<, <=, >, >=, ==, !=` | Comparison |
| `in, not in` | `x in y` check if `x` is contained in the sequence `y` |
| `is, is not` | `x is y` is `True` if `x` and `y` are referencing the same object |

### Logical Operators

Python supports these logical (boolean) operators, that operate on boolean numbers.

| Operator | Description |
|---|---|
| `and` | Logical AND |
| `or` | Logical OR |
| `not` | Logical NOT |

Notes:

- **Python's logical operators are typed out in word, unlike C/C++/Java which uses symbols `&&`, `||` and `!`.**
- There is no exclusive-or (`xor`)