> Python workshop
>
> ***1-1. Introduction***

About the workshop

Workshop objectives

- Understand the basis of programming.
- Learn how to program with Python.
- Be able to learn other programming languages by yourselves.

Plan

Day	Topic	Details
Day 1	*Introduction*	• Language syntax, variables, basic I/O • Control structures
Day 2	*Working with data*	• Data Structures • String and String formatting
Day 3	*Advanced topics*	• Functions and classes • Data manipulation and visualization

Course materials

All materials can be accessed through our Moodle page.

- Links to notes on Google Colab. You can also use the PDF version.
- Files created in demonstration
- Links to recordings

Workshop format

- This workshop is designed to encourage self-learning. For each topic, there are 3 major components:

1. Quick topic introduction and demonstrations.
2. Self-learning materials with self-evaluation exercises**.
3. Extended self-learning materials for future reading.

** For students who needs to pass the workshop, you must complete the self-evaluation exercises (with grades received) on Moodle.

Self-evaluation: Quiz

You must provide an answer that could be recognized by Moodle to receive a grade. For questions that asks for python code, please provide the shortest possible answer (e.g., remove spaces between variables and operators).

You will be able to re-attempt the quiz after submission.

Self-evaluation: VPL exercises

- VPL is a coding environment on Moodle that runs and evaluates your code. In these exercise, your code must provide an **exact** output according to the specific inputs. You need to press the "evaluate" button to check your code. Make sure you have received a grade after evaluation.
  - All **input** to the program must be handled by using `input()`. New test cases may be added after you have submitted your program.
  - The **output** must match exactly with the required output format. E.g., `Hello, world!` is considered a different output from `hello, world!` or `Hello,world!`.
  - You are advised to setup your own environment for you coding practices. The VPL is only for self-evaluation purpose.
- Demo exercises and optional exercises will not be available on Moodle. You are advised to set up your own environment to write/test Python programs.

Programming environment

Installing Python

- In this workshop we will be using the Python version 3.12.
  - Downloadable at https://www.python.org/.
  - Google Colab may use an older version of Python (3.10 as of June 2024), but it doesn't matter as most of the materials run well in Python 3.10+ .

- Once you have installed Python, you should be able to run python in a terminal on your machine. In most cases, the command `python` will be available in the Terminal/Command Prompt/PowerShell after installation. However, depending on the OS you are using, the command could be a bit different:
    - On MacOS, the command could be `python3`
    - On Windows, the command could be `py`

Running Python

Interactive mode

- We may start a Python shell using command `python` (or `python3` / `py`) in a terminal. This allows us to test-run Python code quickly.

Script mode

- To formally write (and run) a Python program. We put our code in a text file (with a `.py` file extension), then run it with the same python command. E.g., If the program is named `your_program.py`, we run the command `python your_program.py` to run it in the terminal.

Notebook format

- The Notebook format (`.ipynb`) provides a way to combine text blocks (written in markdown) and code blocks in a single document. You need to use a compatible editor/platform to view/edit it.

Development environment

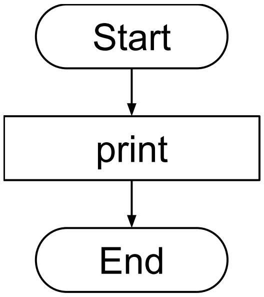
- An integrated development environment (IDE) provides efficient ways to write code and test them in a unified UI.
- **IDLE** is an IDE that comes with Python. It starts with a python shell (for interactive mode), with an option to edit files in a file editor for code writing and testing.
    - We will use IDLE in all demonstrations.

Introduction to Programming

What is a program?

- A program usually defines a **sequence** of statements to be executed one by one.

- In a sequential, synchronous program, a program defines a flow of control that performs a task.



Here is our first program:

In [1]:

```
# Hello, world
print('Hello, world!')
```

```
Hello, world!
```

Hello, world explained

- There are only two lines in our first program:
- The first line is a comment, it documents **what** the program does.

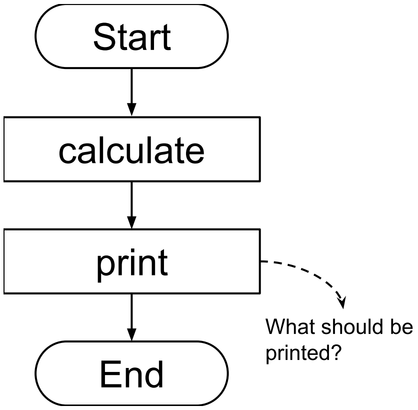
```
# Hello, world
```

- The second line is an **output** statement, it prints the string `Hello, world!` to the console.

```
print('Hello, world!')
```

Concepts: Program state

- The previous program is not very useful. We hope to produce a more interesting output. But how is that possible?



Variables and I/O

- To allow statements in a program to work collectively, we need to maintain a **program state**. A statement in a program produces result depending on the current program state, in the process, the program state may be changed.

- We use **variables** to maintain such state.

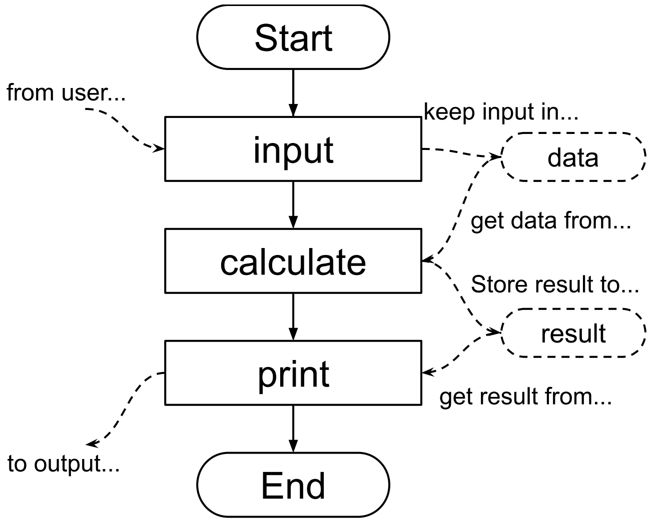
    - Program can accept inputs to alter the state.
    - Program can change the state by results of computation.
    - Program can output information according to the state.

Concepts: variables and I/O

- **Variables**: maintain program state

- **Input**: collect information from user/external

- **Output**: provide information to user/external



Example

In [2]:

```
mystr = input('Please input your name:')
print('Hello', mystr)
```

```
Please input your name: K
Hello K
```

- `input()` is used to ask user for an input, in the form of a string.
- `=` operator is used to assign the result of `input()` to a variable named `mystr`.
- `print()` is used to print some text.

Another example

In [3]:

```
a = float(input())
b = 2 * a + 1
print(a, b)
```

```
 1
1.0 3.0
```

- We use `float(input())` to read a floating point value from user. Note that the input prompt is omitted.
- It is possible to use basic arithmetic operations in Python.

Quick Quiz

What is the output of the following code?

```
a = 34
b = 80
```

```
print("a", "+", b, "=", a + b)
```

A	B	C	D
`a + b = a + b`	`34 + b = 114`	`a + 80 = 114`	`34 + 80 = 114`

Demonstration 1-1

- Referring to the previous examples, write a program that reads **two floating-point values** and compute their harmonic mean.
- Harmonic mean H of input values a, and b can be calculated by the formula $\frac{1}{H} = \frac{1}{2}(\frac{1}{a} + \frac{1}{b})$, or simply $H = \frac{2ab}{a+b}$.
- Sample input/output:

Input	Output
1 4	1.6
3 7	4.2
3.7 4.3	3.9775

Self-learning topics

Variable, operators and basic I/O

- Values and variables
- Output options
- Arithmetic operators
- Reading input from user

Values and Variables

- To maintain program states, we associate values to names.
- These names are called variables.
- In Python, we use the **assignment operator** `=` to assign a value to a variable, for example:

```
a = 10
```

- In many programming language, variable must be **declared** before it could be used.
  - In Python, assigning a value to a name automatically declare the variable.

- The **assignment operator** `=` will always assign the **value** on the right to the **variable** on the left.
- In the above example, variable `a` is assigned a value of 10.

Using variables

If we assign a new value to a variable, the variable will be overwritten. For example:

```
a = 10
a = 20
```

In the above example, variable `a` is assigned a value of 20.

Types of values

A variable can be used to hold different types of values. Typical variable type in Python are **integer**, **floating-point**, and **string**.

```
# integer/floating-point value
a = 3
b = 1.5
# string (same for single quote or double quote)
c = "hello"
d = 'world'
```

Boolean values

Python also provide **boolean** values `True` and `False`. Their values are equivalent to integer `1` and `0` respectively.

```
# boolean values
e = True    # equals 1
f = False   # equals 0
```

Output options

It is common to print a combination of values and variables. Consider the program below, the output is not desirable (try it!). Sometimes, we pefer printing values on the same line.

In [4]:

```
a = 1
b = 2
c = 3
print(a)
print('+')
print(b)
print('=')
print(c)
```

```
1
+
2
=
3
```

Output multiple values

We can print a list of values, separated by **commas** when we use `print()`, a space will be added between the values:

In [5]:

```
a = 1
b = 2
c = 3
print(a, '+', b, '=', c)
```

```
1 + 2 = 3
```

Separator option: `sep`

We can specify the **`sep`** option in `print()` to specify the separator to be used when printing multiple values.

In [6]:

```
print(1, 2, 3)
print(1, 2, 3, sep=',')
```

```
1 2 3
1,2,3
```

Ending option: `end`

Another option for `print()` is **`end`**, which control how to end the printing. By default a **new line** is inserted.

In [7]:

```
print(1, end=' + ')
print(2, end=' = ')
print(3)
```

```
1 + 2 = 3
```

Sometimes it will be usful to end with an empty string. For example, when we want to output a single line with multiple print statements:

In [8]:

```
print(1, end='')
print(2, end='')
print(3)
```

```
123
```

Arithmetic operators

For integer and floating point numbers, we can use the four **arithmetic operators**: `+`, `-`, `*` (multiply), and `/` (divide). For example:

```
In [9]:  print(1 + 2)
```

```
3
```

```
In [10]: print(10 - 1.5)
```

```
8.5
```

```
In [11]: print(2 * 6)
```

```
12
```

```
In [12]: print(9 / 6)
```

```
1.5
```

Order of execution (operator precedence)

When there are multiple operators in the same statement, the order of execution follows a certain rules. For the basic arithmetic operators, it follows the basic mathematics rules (multiplication and division first), and is processed left-to-right.

```
In [13]: print(1 + 2 * (3 - 4) / 5)
```

```
0.6
```

If unsure, always add brackets to specify the order of execution.

Reference: https://docs.python.org/3/reference/expressions.html#operator-precedence

Floating point division and floor division

Operator `/` always results in a **floating point number**, even when both operands are integers.

```
In [14]: a = 100
         b = 10
         print(a / b)
```

```
10.0
```

If **integer division** is needed, we can use the floor division operator (`//`) instead. This operator will perform division and return the floor of the result. For example:

```
In [15]: print(9 // 6)
```

```
1
```

```
In [16]: print(100 // 10)
```

```
10
```

Power operator

The **power operator** `**` calculates and returns the value of a base raised to a specific power.

Note: the `^` operator in Python is another operator, which will be discussed later (optional).

In [17]:

```python
print(10 ** 2)
```

```
100
```

In [18]:

```python
print(2 ** 10)
```

```
1024
```

Modulo operator

The **modulo** operator `%` calculates and returns the **remainder** of dividing first operand by the second operand.

The two operators, `//` and `%` can be used to find the quotient and remainer of a division operation.

If, `q = a // b` and `r = a % b`, then `a = b * q + r`.

This operator is extremely important in computer science. (why?)

In [19]:

```python
print(100 % 3)
```

```
1
```

In [20]:

```python
print(100 % 7)
```

```
2
```

Reading input

The `input()` function will always read a string from user.

In [21]:

```python
a = input('Please input a string:')
print(a)
```

```
Please input a string: Hi!
Hi!
```

The part `'Please input a string:'` is a message to prompt user for input. It can be omitted:

In [22]:

```python
a = input()
print(a)
```

```
 Hi!
```

```
Hi!
```

Reading integer or floating-point values

As `input()` will always return a string, before we can use an input value in arithmetic calculation, we need to convert it to integer or floating point values. For example:

In [23]:

```python
a = int(input())
b = float(input())
print(a, '+', b, '=', a + b)
```

```
 1
 2.3
1 + 2.3 = 3.3
```

Self-evaluation exercises (1-1)

Quiz

- You can answer these questions on Moodle.
- Moodle expect exact answers, please remove all spaces if the question asks for code snippet.

Programming Exercise

- You can attempt these exercises on VPL (on Moodle).
- When doing exercises on VPL, you must follow the exact input/output requirement. You must also press the "evaluate" button to evaluate your work for it.

Quiz 1-1

1. Name the operator `//`.
2. What is the output of the following program? Try to derive the output without running the code.

```python
print('Hello', 'oh', sep=',', end=' ')
print('my', 'world', sep='', end='!')
```

Exercise 1-1

- Write a program that convert time period (in seconds) to the long format represented by the pattern `?h ?m ?s`.
- Sample input/output:

Input	Output
100	0h 1m 40s

Input	Output
10000	2h 46m 40s
1000000	277h 46m 40s

Optional topics

These topics are optional. You are encourage to read them when you have time to understand more about the Python programming language.

- Bitwise operators
- Type hint

Bitwise operators

Number base

Apart from base 10 numbers, we can define numbers with base 2, 8 and 16 in Python.

- Base 2 number is prefixed by pattern `0b`.
- Base 8 number is prefixed by pattern `0o`.
- Base 16 number is prefixed by pattern `0x`.

In [24]:

```
print(0b011010100)
```

```
212
```

In [25]:

```
print(0o324)
```

```
212
```

In [26]:

```
print(0xD4)
```

```
212
```

Binary representation

- In a computer, all values are stored as binary numbers.
- So number `212` is internally stored as `0b11010100`.
- These numbers are left-padded with zeros to match with the bit-length. Therefore the number `212` is actually stored as `0b00...011010100`.
- Note that in a signed representation, numbers are stored in 2's complement representation.

Bitwise operations

Bitwise operations apply on numbers bit by bit, for example, the AND operation (`&`) on values 12 (`0b01100`) and 10 (`0b01010`) will be:

```
00...01100 (12)
& 00...01010 (10)
------------------
  00...01000 ( 8)
```

Only one of the bits above will give a result of 1 as both operands are 1.

Bitwise logical operators

AND / OR / XOR

Bitwise logical operators includes AND (`&`), OR (`|`), and XOR (`^`), for example:

In [27]:

```python
a = 0b01100
b = 0b01010
print(a & b, a | b, a ^ b)
```

```
8 14 6
```

NOT

There is also the negation operator (`~`) which inverts all the bits. For example, positive value `01100` (`00...01100`) will becomes `11...10011` .

In 2's complement representation. The above value equals `-13` .

In [28]:

```python
a = 0b01100
b = 0b01010
print(~a, ~b)
```

```
-13 -11
```

Shift Operators

Shift operators shift the binary pattern to the left (`<<`) or right (`>>`). For example:

In [29]:

```python
x = 0b01101
print(x >> 2, x << 1)
```

```
3 26
```

`x >> 2` shifts value `0b00...01101` (13) two positions to the right, therefore the result is `0b00...011` (3). Similarly, `x << 1` shifts the same value one position to the left, so the result is `0b00...011010` (26).

Type hint

Every value has a "type"

We can check the type of a value using `type()`.

In [30]:

```
print(type(1))
print(type(3.14))
print(type('Hello'))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
```

Variable type

When we assign a value to a variable, the type of variable changes accordingly.

In [31]:

```
a = 1
print(type(a))
a = 3.14
print(type(a))
a = 'Hello'
print(type(a))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
```

Type hint

It is always advised to keep the type of a variable unchanged. In many programming languages, you have to declare the type for a variable before you can use it. In Python, it is not necessary but possible.

In [32]:

```
myInt: int = 1
myFloat: float = 3.14
myStr: str = 'Hello'
print(type(myInt), type(myFloat), type(myStr))
```

```
<class 'int'> <class 'float'> <class 'str'>
```

But it's just a "hint"

Type hint has no effect during code execution, it is a hint for developers and for the IDE to detect possible error in your program. Variable type will still be changed if you assign value of diffeent type to it.

In [33]:

```
myInt: int = 1
myInt = 3.14
print(type(myInt))
```

```
<class 'float'>
```