Python Fundamentals

- We will need to print the output of any code to have a solid understanding of what is happening behind the scenes to catch problems

In Python, we would use the `print()` function to do so:

```
print("Hello World")
```

Data Types

Strings

Strings are sequence of characters enclosed with either single inverted commas ('') or double inverted commas ("") These are the examples of strings:

```
'Hello World'` `'123'
```

We will not be able to carry out any calculations when you **define numbers as a string**

Integers

Integers are whole numbers **without decimal points** which can be either positive or negative.

These are examples of integers:

```
123` `-456
```

Lists

Lists are a collection of items that are ordered (accessible) and mutable (changeable). Lists can contain elements of different data types and its elements are enclosed in [] square brackets separated by commas.

These are the examples for lists:

```
[1, 2, 3]` `[ABC, 88, DEF, 99]
```

Dictionaries

Dictionaries are a collection of key-value pairs similar to the real world dictionary. The `key` is similar to the unique `word` in a dictionary and the `value` represents the meaning of the word. Dictionaries are defined with curly braces {} with its different keys separated by commas.

Here are some examples of a dictionary:

```
{"A" : 99, "B": 55, "C": 72}` `{"Singaporeans" : 50, "Malaysians": 50 }
```

Booleans

Booleans represent either true or false values. If we are converting an integer to a boolean, any non-zero number will be evaluated to `True` and zero values would be evaluated to `False`. These are the booleans in Python (take note of capitalisation!):

```
True  False
```

Why Variables?

We can assign the values of different data types into a variable to **prevent repeating ourselves**. For example, if we are planning to use our full name many times in our code, we can assign it to a variable `name` and use it.

Rules for Defining Variables

- Variable names **cannot start with numbers or special characters excluding underscores**.

- You cannot use **spaces** when defining the variable and would need to use `_` to replace it.
- Use meaningful variable names for others to understand. For example, do not use the variable name `age` to represent a person's name.
- Variables are case sensitive. `Name` and `name` are treated as different variables.

Examples of Variable Names

Valid:
`C0mputing`
`i_love_computing`
`_computing`

Invalid:
`1love`
`i love computing`

Defining Variables:

To assign a value to a variable, we use the `=` sign.
`name = "Computing Kid"` (string)
`age = 88` (integer)
`hobbies = ["coding", "eating", "sleeping"]` (list)
`adult = True` (boolean)

Printing with Variables

From Python 3.6 (A-Level Version), we can use `print(f'')` to concisely combine variables together. Let us define some variables and use `print(f'')` to print all of them in one statement:

```
name = "Computing Pro"
age = 88
hobbies = ['lim kopi', 'jiak beng']
print(f"Hi, I am {name} and am now {age}. My hobbies are {hobbies}")
```

Output: `Hi, I am Computing Pro and am now 88. My hobbies are ['lim kopi', 'jiak peng']`

Hmm... The hobbies are still in the list format! We will solve this in the section of `CRUD with Lists`.

1. Creating a List

We can initialize a list with square brackets ([]) with all of the elements separated by commas. Let's define a list named hobbies:

```
hobbies = ['basketball', 'badminton', 'bowling']
```

2. Zero-Based Indexing

You can think that all of the elements are given index numbers for us to easily access the elements. The first element is given the **index of zero!** instead of one.

3. Reading a List

To access the first element of the list, we will access it with its variable name followed by square brackets containing its index:

```
print(hobbies[0])
```

Output: `basketball`

To access the last element of the list:

```
print(hobbies[2])
```

Output: `bowling`

We can also **use negative indexing** to access elements from the back of the list. The last element of the list is given the value of `-1`.

To access the last element of the list with negative indexing:

`print(hobbies[-1])`

Output: `bowling`

4. Updating a List

4a. Append: Add element to the back of the list

We can add an element at the end of the list by using the `.append` method. To add a new hobby in the `hobbies` list:

`hobbies.append('skating')`

Now when we print out hobbies:

`print(hobbies)`

Output: `['basketball', 'badminton', 'bowling', 'skating']`

4b. Insert: Add an element into our list with an arbitrary index

We can apply the `.insert()` method with two arguments: `index` and `value`. We can add skating to the front of our list in this manner:

```
hobbies = ['basketball', 'badminton', 'bowling']
hobbies.insert(0, 'skating')
print(hobbies)
```

Output: `['skating', 'basketball', 'badminton', 'bowling']`

5. Delete

5a. Pop - Remove last element of the list

We can use the `pop` method in order to remove the last element of the list. Given a list hobbies:

```
hobbies = ['basketball', 'badminton', 'bowling']
hobbies.pop()
print(hobbies)
```

Output: `['basketball', 'badminton']`

5b. Remove - Remove an element based on its value

Given a known value, we can remove that element with the `.remove()` method:

```
hobbies = ['basketball', 'badminton', 'bowling']
hobbies.remove('basketball')
print(hobbies)
```

Output: `['badminton', 'bowling]`

5c. del - Remove an element based on its index

Given a known index, we can remove that element with the `del` keyword in this format. For example, to remove the second element of `hobbies` below:

```
hobbies = ['basketball', 'badminton', 'bowling']
del hobbies[1] # Recall Zero-Based Indexing
print(hobbies)
```

Output: `['basketball', 'bowling']`

Condition Testing

We can control what the next step in any program with conditions which will evaluate to either `True` or `False` (boolean values). Here is how we test conditions in Python:

`>:Greater`

`>=` Greater than or equal to

< : Lesser than

<= : Lesser than or equal to

!= : Not equal to

== : Equal to

Let's print the output of the following conditions:

```
print(20 == 40)
print(20 != 40)
print(20 > 40)
print(20 < 40)
```

Output:

```
False
True
False
True
```

If Statements

Here is the syntax of using `if` statements:

```
if condition:
    ...code...
```

Notice the indentation (either 2 or 4 spaces) after the colon

Let us try an example by testing if the person age is greater than 18:

```
age = 18
if age >=18:
    print("You are of age!")
```

Output: `You are of age!`

If-Else Statements

We can also use `else` to execute some other code when none of the conditions in the control block is fulfilled:

```
age = 18
if age >=18:
    print("You are of age!")
else:
    print("You are not of age to enter")
```

If-Elif-Else Statements

If we can test multiple conditions with the `elif` statements if the first condition is not fulfilled. We will be printing an error when the age inputted is negative here and **it is used in between `if` and `else`**:

```
age = -5
if age >=18:
    print("You are of age!")
elif age <= 0:
    print("You entered an invalid age")
else:
    print("You are not of age to enter")
```

Output: `You entered an invalid age`

For Loops

Some tasks are repetitive in programming, and a loop will help us do a task until a condition is fulfilled. Sometimes, we will also want to iterate through our lists & dictionaries.

For Loop Syntax

```
for variable in iterable:
    Execute this block of code
```

`Variable` can be named anything you want, but the common names are `i`, `x`, or `_`.

`Iterable` are objects which can return its member one at a time (e.g., strings, lists, dictionaries).

Given a list of fruits, we can print out every element of the list with a `for` loop:

```
fruits = ['apple', 'banana', 'cherry']
for fruit in fruits:
    print(fruit)
```

Output:

```
apple
banana
cherry
```

We can also loop through a dictionary (which contains key-value pairs) with a slightly modified for loop:

```
grades = {'A': 50, 'B': 20}
for key, value in grades.items():
    print(f"The amount of people who got {key} is {value}")
```

```
The amount of people who got A is 50
The amount of people who got B is 20
```

Here we specify **2** arbitrary variables to represent the `key` and `value` for every iteration of the loop. In the first iteration of the loop, `key` holds the value `A` while `value` holds the value `50`. In the second iteration of the loop, `key` holds the value of `B` while `value` holds the value of `20`.

While Loop

If we want to run some code until a condition is fulfilled, we can use a `while` loop.

Syntax

```
counter = 1
while counter < 5:
    print(counter)
    counter += 1
```

Output:

```
1
2
3
4
```

Functions

When we need to perform the same task over and over again, we can reduce duplication in our code base with functions. To define a function, we can use the `def` keyword and pass in parameters which need to be used in the function.

Syntax:

```
def function_name(param1, param2):
    <code here>
    return <variable>
```

For example, we can write a function called `add_ten` which adds 10 to a number passed to it:

```
def add_ten(number):
    return number + 10
```

We can now use the function and print the result in this manner:

```
print(add_ten(5))
```

Output: `15`