[Session 1] Python Basics Supplement

`print()` function

`print()` function is one of most important function in Python, because we can see the result of expression or function's return value by using this.

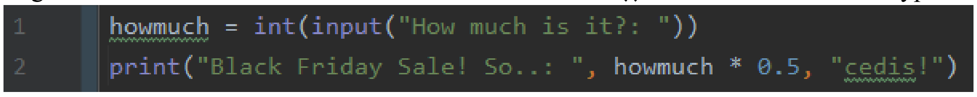
Basically, we can use like this:

- ➢ **`print(value, sep, end)`**
- `value`: Something that can be printed. As we mentioned in the lecture, this can contain multiple items.
- `sep`: We can specify the separator. If we print multiple items, then we put separator between each item. For example, `print(1, 2, 3, sep="_")` will print `"1_2_3"`. The default is " ", a whitespace.
- `end`: We can specify ending character. The default is "\n", a line feed. (and this is why many lines are printed when we call `print()` function many times) For example, `print("hello", end="!")` prints `"hello!"`, without line feed.
- Note that, these can be omitted all, `print()` is also ok.

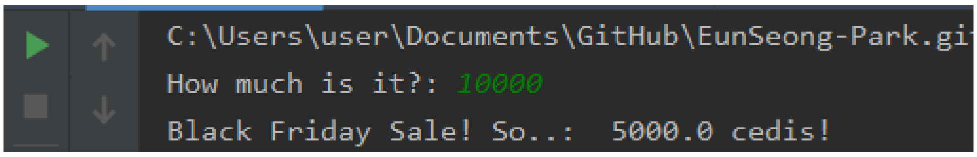
`input()` function

We can get some input from user, by using **`input()`** function. We can show some prompt(the signal or message that computer is ready for the input) with parameter. (Like `print()` function, we can put many things as long as these can be printed)

The return value of the `input()` function is the **string** you are typed (and then Enter). Sometimes, we may want to get a number from user. How can we do that? Use `int()` function to convert its type.



```
howmuch = int(input("How much is it?: "))
print("Black Friday Sale! So..: ", howmuch * 0.5, "cedis!")
```



```
C:\Users\user\Documents\GitHub\EunSeong-Park.gi
How much is it?: 10000
Black Friday Sale! So..:  5000.0 cedis!
```

Special characters

String can contain some special characters.

- '\n': line feed.
- '\b': backspace
- '\t': tab
- '\\': backslash(\)

Boolean

We can apply many operations between boolean values.

- **`A and B`**: if A and B are True, then True. Otherwise False.
- **`A or B`**: if either A or B are True, then True.
- **`not A`**: if A is True then False. if A is False, then True.

Also, we can get boolean in many ways:
- **`!=, ==, >, <, >=, <=`**: If the result is true, then True.
- **`A in B`**: For an iterable object (such as string, list) B, if A is included in B, then True. Otherwise False. (You also can use `not in`)
- **`A is B`**: If A and B are same object with common ID, then True. Otherwise False. (You can see each id by using `id()` function)

There are some values that can be evaluated to False.
- `False`
- `0`
- `None`
- `[], (), "", ''`

Indexing & slicing of list / tuple / string

These three datatypes have in the following commons:
- **Can be iterated**: We can traverse each element (or character). (e.g. `for` statement)
- **Can be indexed:** We can access ith element directly, with [i]. (e.g. `(1, 3, 2, 1)[2]` => `2`)
- **Can be sliced:** We can get a sub-data from the original, according to certain rules

Indexing and slicing are very important (and powerful) operations in Python. You should be accustomed to using these.

(Index Rule)
1. **Index starts at 0**. In order to access to ith element, we use `(Data)[i-1]`, not `(Data)[i]`.
2. Index out of range will cause error. For example, `(1, 2, 3)[3]` is an invalid indexing
3. Negative index can be valid. For example, `list[-1]` is the last element, and `list[-2]` is the second-last element. Note that, if the number of elements is n, then index under -n is invalid.
   - So, For some list with n items, `list[i] = list[-(n-i)]`

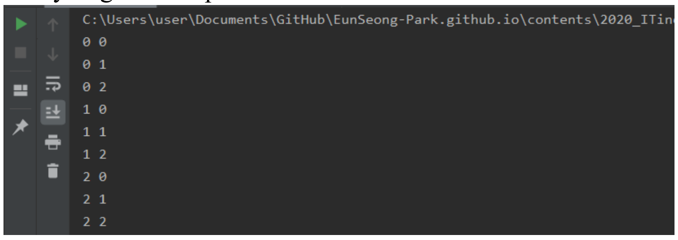
(Slicing Rule)
1. Basically, for some iterable data, A, use `A[start:end:step]`. All of them can be omitted. (even A[:] or A[::] are ok) The default of step is 1. For a list A, with n items
   A. `A[:end] = A[0:end] = A[0:end:1]`
   B. `A[start:] = A[start:n-1] = A[start:n-1:1]`
   C. `A[:] = A[::] = A`
2. In `A[start:end]`, `A[end]` is not included.

We uploaded some exercises for practice of this.

Nested loop

Some loop can be in another loop. We call this **nested loop**. Let's see the example.

```
for i in range(3):
    for j in range(3):
        print(i, j)
```
Can you guess the print result?



```
C:\Users\user\Documents\GitHub\EunSeong-Park.github.io\contents\2020_ITin
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
```

More about loop

You can use loop (for / while) in many ways. Some keywords are provided.

- **`pass`**: Do not anything.
- **`continue`**: Complete the current iteration immediately, and then start next one (if possible)
- **`break`**: Terminate the entire loop immediately.

`pass` can be used in other statements using colon(`:`) such as `if`/`elif`/`else`, `def` (function). Usually, we can use it as a placeholder. Note that if there is no content after colon-statement, then an error is occurred.

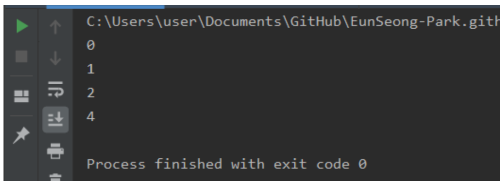
```
if False:
else:
    print("hello")
```
It should be like this:

```
if False:
    pass
else:
    print("hello")
```
Or, in `for` statement,

```
for i in range(100):
    pass
```

We sometimes want to skip an iteration or skip the following codes. Then we can use **`continue`** keyword. We can use it like this:

```
for i in range(5):
    if i == 3:
        continue
    print(i)
```
In this example, numbers between 0 and 4 are printed, except 3 because `print(i)` was not executed when `i` was 3.



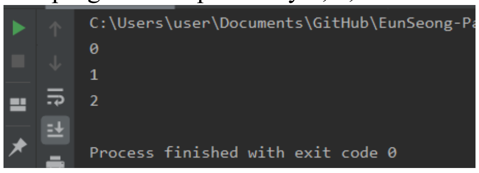
```
C:\Users\user\Documents\GitHub\EunSeong-Park.gith
0
1
2
4

Process finished with exit code 0
```

`break` terminates the entire loop. For example,

```
for i in range(1000000):
    if i == 3:
        break
    print(i)
```

The program will print only 0, 1, 2. because the loop was broken when i was 3.



```
C:\Users\user\Documents\GitHub\EunSeong-P
0
1
2

Process finished with exit code 0
```

Note that, `break` and `continue` breaks / continues ONLY ONE nearest loop.

```python
my_sum = 0
for i in range(3):
    for j in range(3):
        my_sum += 1

print(my_sum)
```

This program prints 9. How about this?

```python
my_sum = 0
for i in range(3):
    for j in range(3):
        if j == 2:
            break
        my_sum += 1

print(my_sum)
```

Since the `break` does not break the i-loop, the program will print 6.

Function: recursion

The situation that some function calls itself is called "**recursion**". Recursion is sometimes useful, because it can simplify the code. For example, we will implement "Fibonacci number" function in two ways.

```python
def fibo(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        result = 0
        a, b = 0, 1
        for i in range(2,n+1):
            result = a + b
            a = b
            b = result
        return result
```

```python
def fibo_recursion(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibo_recursion(n-2) + fibo_recursion(n-1)
```

The latter code has better readability and is more straightforward.

However, note that, **recursion must be ended eventually**. If the function calls itself infinitely, our computer cannot withstand and causes some problem. This is called "**stack overflow**". (Fortunately, in most programming language prevents this, but we still cannot run the program properly.) So read your code carefully, and ensure that the recursion can be terminated eventually.

Function: parameter and argument

In the lecture, we used the terms "parameter" and "argument". How do they differ? Usually, we define these like:

- **Parameter**: The name of "variable" when some function or method is defined
- **Argument**: The "value" passed to some function or method

So, for example,

```python
def one_adder(n):
    return n + 1


one_adder(15)
```

We can say that, the function, `one_adder` uses a parameter, `n`, and we called `one_adder` with a argument, `15`.