Python, 파이썬

참고자료: https://docs.python.org/2/tutorial/

1. 연산자와 변수 Operator and variable

1.1 연산자

화씨온도를 섭씨온도로 변환하는 경우,

Temperature_C = (Temperature_F – 32) x 9/5

<표 1.1> Operators in Python

+, -, *, /	다하기, 빼기, 곱하기, 나누기
%	나머지 (modulo)
//	몫
**	제곱

1 Computer에서 수(number or numeric value)를 표현하는 방법은 크게 정수형(integer type)과 실수형(float type)으로 구분된다.

2 Interactive mode에서 Python 명령문 각 line은 ≫ (prompt or cursor), ... (연속), 그리고 #(각주, comment)로 시작된다.

3 변수의 형태나 연산 결과는 자료형 반환 함수 'type(argument)'를 사용하여 점검할 수 있다.

≫ type(10)

≫ type(10.0)

Example

```
≫ 2+2              #4 (int)
≫ type(2+2)        #<class 'int'>
≫ 50 – 5*6         #20 (int)
≫ 4*3.75 – 1       #14.0 (float): 실수형 수가 연산에 포함되면 결과는 항상 실수형
≫ 8/5              #1.6 (float)
≫ 8/4              #2.0 (float): Division (/) 연산은 항상 실수형 값을 출력한다.
≫ 17/3             #5.666666666666667 (float)
≫ 26//2            #13 (int): Floor division, 몫: 소수점 이하를 버린다.
≫ 31//4            #7 (int)
≫ 31%4             #3 (int): Remainder, 나머지, 31 = (31//4)*4+3
≫ 2**4             #16 (int): Power, 2^4
```

연산자 결합법칙 Operator Associativity, Operator Precedence Rule

연산식에서 하나 이상의 연산자가 나타날 경우, 연산자 간의 우선 순위법칙에 따라 연산한다.

1. 괄호 – 지수 – 곱셈, 나눗셈, 나머지, 몫 – 더하기, 빼기
2. 같은 부류의 연산자가 연결된 경우에는 왼쪽에서 오른쪽으로 (단, 지수의 경우, 오른쪽에서 왼쪽으로)

Example

```
≫ (2 + 4) * 5
≫ 4 * (26 – 9)
≫ 4 * 6 – 5
≫ 9 + 54 / 6
≫ 2 ** 4 + 5
≫ 6 * 5 ** 2
≫ 8.2 * 4 + 2
≫ 20 + 9 / 4.5
≫ 5 + 12 // 7 * 3
≫ 15 / 2 + 21 % 4 – 2 ** 3
```

1.2 변수, Variable

(1) 변수명 규칙

프로그램 과정에서 변수를 설정할 때, 가능한 변수의 기능을 예측할 수 있는 형태로 구성한다.

1. 변수의 이름은 문자, 숫자, 그리고 _(underscore)만을 포함할 수 있다 (특수문자 사용 불가).
2. 변수명은 문자 또는 _(underscore)로 시작할 수 있다.
3. 미리 지정된 단어를 사용할 수 없다: 예약어와 지정단어(preserved word or keyword)는 'input keyword / keyword.kwlist 명령문으로 확인할 수 있다.
4. Case sensitive

(2) 변수의 할당, Assignment Statement

할당문 연산자, =

Example

```
≫ number
≫ number = 5        # interactive mode에서는 변수 number에 값 5가 할당되지만 결과는
                    # 표시되지 않는다.
≫ number_1 = 26
≫ number_2 = 2 * 5 ** 2
≫ number_3 = number + 2
≫ number_1 = number_1 + 1        #복합대입 연산자 표현, number_1 += 1, number_2 *=
≫ number_1 = number_1 * 2        #number_1 *= 2
≫ number_1 = number_1 // 2       #number_1 //= 2
```

2. 자료의 입출력, Input/Output of variables

2.1 자료형, Variable Class

(1) 정수와 실수, 'int' and 'float'

정수형은 소숫점을 갖지 않는 data type이며, float는 소숫점을 갖는 data type이다.

```
≫ int(3.5)          # 3
≫ 2e3               # 2000.0
≫ float("1.6")      # 1.6
≫ a = None          # 'None'은 아무런 data를 갖지 않는다는 것을 표현한다.
≫ v = 2 + 3j        # 복소수의 표현식
≫ v.real            # 2
≫ v.imag            # 3
```

(2) bool type

bool type은 'True/False (참/거짓)' 값만을 갖는 형태이다.

```
≫ bool(0)           # False
≫ bool(-1)          # True
≫ bool('False')     # True
```

(3) 문자, string class

Python은 숫자 뿐만이 아니라, 문자열(string)도 처리할 수 있다. 문자열은 single quote('...') 혹은 double quote("...")로 입력된다. 문자열 변수의 정의도 동일하게 이루어 진다.

```
≫ 'I am a student'
≫ 'I don₩'t know'   # string 내부에 (') 혹은 (")가 포함되어 있으면 '₩'를 사용하여 string의
                    # 일부임을 표시
≫ "I don't know"
≫ '"Yes", he replied'
≫ str_1 = 'I am a boy'
```

(4) String 연산자

병합, concatenation, +

```
≫ last_name = 'Hong'
≫ first_name = ' Gil Dong'
≫ 'Hong' 'Gil Dong'
≫ full_name = last_name + first_name
≫ last_name + 'Dong'
```

반복, iteration, *

```
≫ 3*'un' +'ium'             # 'ununununium'
≫ laughing = 'ha' * 3
```

≫ start = '=' * 10
≫ title = 'Python Program'
≫ print(start + title + start)

String 길이, len

≫ size = len('I am a boy')

String indexing

String은 행렬에서처럼 각 성분을 index(or subscript)로 구분할 수 있다. 첫 번째 글자(character)는 index 값이 0이다.

≫ san = 'Mountain'
≫ san[0]
≫ san[3]
≫ san[8] #'len(san)' returns 8.
≫ san[-1] #'n': last character
≫ san[-2] #'i': 2nd last character

변수는 처음에 값을 할당 받은 후, 다른 값으로 변경이 가능하다. 하지만, string 형은 한번 지정하면 index를 이용한 변경을 허용하지 않는다 (immutable).

≫ san[2] = 'a'

String의 일부분 추출, slice

<표 2.1> string의 slice 범위 지정

[:n]	처음부터 n을 포함하지 않는 범위까지 지정하여 slice
[m:]	m번째부터 string의 끝까지 범위를 지정해서 slice. 단, m이 문자열의 길이보다 긴 경우에는 null string을 지정

≫ san[0:2] #'Mou': 1st three characters
≫ san[2:5] #'unt'
≫ san[:2] #same as 'san[0:1]': index-2 character는 포함되지 않는다.
≫ san[4:] #same as 'san[4:7]': index-4 character는 포함된다.
≫ san[:4] + san[4:] #'Mountain': same as 'san' itself.

2.2 입력과 출력

input(), print()

≫ name = input()
≫ name = input('Enter your first name.')
≫ age = input('How old are you?')
≫ type(age)
≫ print('I love you.')

```
≫ math = 26
≫ english = 31
≫ physics = 96
≫ print(math, english, physics)
≫ print('Total sum is₩n', math + english + physics)
```

3. 조건문, if statement

3.1 Boolean 형과 비교/논리 연산자

'강의에 4회 이상 결석하면 F학점이다'라는 조건을 고려해 보자. 이 조건에 대하여 '참/거짓(True/False)으로 답할 수 있으며, 각각에 맞게 지정된 작업을 수행할 수 있다. 이때 True/False 값을 Boolean 값이라 부르고, True/False 값을 저장하는 변수를 Boolean형 변수라 한다.

```
≫ a = True
≫ type(a)
≫ a
```

비교 연산자

== (equal), != (not equal), > (greater than), < (less than), >=, <=

```
≫ n = 3
≫ n == 3          #True
≫ n == 5          #False
```

Example: Fibonacci 수열을 계산하는 Python program을 작성해 보자

```
≫ a, b = 0, 1          # multiple assignment
... while a<10:        # 'while-loop'는 조건 'a<10'이 만족되는 한 계속 수행된다.
...     print(a)       # 다음 두 line은 'while-loop'의 body를 구성한다. Body를 구성하는
...     a, b = b, a+b  # line은 tab을 사용하여 indent하여 작성하여야 한다.
...
```

3.2 조건문, Conditional Statement

Python의 조건문은 if라는 keyword로 시작한다. 'if' 조건문은 하나의 절(phrase)로 기술하는데, 조건을 명시하는 '조건부(condition)'와 수행할 작업을 명시하는 '수행부(action)'로 구성되어 있다. Boolean식으로 구성되는 조건식이 참(True)이 되는 경우에만 작업이 수행된다.

if statement format

```
if condition:
        True statements
Next statements
```

1 조건이 true이면 'True statements'를 실행하고, 다음에 'Next statements'를 실행

2 조건이 false이면 'Next statements'를 실행

Example: 하나에 1,000원인 연필과 하나에 2,000원인 펜이 있다. 구입 시, 총액이 10,000원을 넘으면 10% 할인해 준다고 할 때, 연필과 펜의 숫자를 입력 받아, 총액을 계산하는 program을 구성하자.

논리 흐름도

sum = (1000 * pencil + 2000 * pen)

if (sum >= 10000), then 10% discount: (i.e., sum = sum * 0.9)

Program

```
num_pencil = int(input('Pencil 개수 입력: '))
num_pen = int(input('Pen 개수 입력))
sum = 1000 * num_pencil + 2000 * num_pen
if sum >= 10000:
    sum *= 0.9
    print("10% 할인 되었습니다.")
print("총합: ", int(sum), "원")
```

논리 연산자

and, or, not

버스 요금은 19세 이상의 일반인은 1300원, 13~18세의 청소년은 1100원, 그리고 6~12세의 어린이는 700원이다. 조건식을 비교 연산자를 사용하여 표시하면 다음과 같다.

(나이 >=6) and (나이 < 13), (나이 >=13) and (나이 < 19), (나이 >= 19)

```
if (age >= 6) and (age < 13):
    passenger_type = '어린이'
    fare = 700
if (age >= 13) and (age < 19):
    passenger_type = '청소년'
    fare = 1100
if (age >= 19):
    passenger_type = '일반인'
    fare = 1300
```

3.3 if-else 조건문

경우에 따라서는 한 조건에 대하여 조건이 참인 경우에 수행하는 작업과 거짓인 경우에 수행하는 작업이 따로 필요한 경우가 있다. 이러한 경우에, if-else 조건문을 사용한다.

if-else statement format

```
if condition:
        True statements
else:
        False statements
Next statements
```

1 조건이 true이면 'True statements'를 실행하고, false이면 'False statements'를 실행한 후, 다음에 'Next statements'를 실행

Example: 백화점 의류 매장에서 t-shirt(10,000원)와 sweater(30,000원)을 구매 금액에 따라 할인해 주고 있다. 10만원 이하 구매 시, 구매 금액의 5% 할인, 10만원 초과 구매 시에는 구매 금액의 15%를 할인해 준다고 한다. 총 금액을 계산하자.

Program

```
num_tshirt = int(input("T-shirt 개수 입력: "))
num_sweater = int(input("Sweater 개수 입력: ")
sum = 10000 * num_tshirt + 30000 * num_sweater
if sum <= 100000:
    sum = sum * 0.95
else
    sum = sum * 0.85
print("합계: ", int(sum), "원")
```

3.4 Nested if 조건문

Example: 0이 아닌 두 숫자를 곱하고, 그 결과가 양수인지 음수인지를 판단하는 program을 작성하자.

Program

```
number_1 = int(input('Enter first number: '))
number_2 = int(input('Enter second number: '))
if number_1 > 0:
    if number_2 > 0:
        print('Positive')
    else:
        print('Negative')
else:
    if number_2 > 0:
        print('Negative')
    else:
        print('Positive')
```

3.5 if-elif-else 조건문

if-elif-else statement format

```
if condition_1:
        True statements 1
elif condition_2:
        True statements 2
else:
        False statements
Next statements
```

1 Condition_1이 true이면 'True statements 1'을, condition_2가 true이면 'True statements b'를 실행하고, 두 조건이 모두 false이면 'False statements'를 실행한 다음, 'Next statements'를 실행

Example: 이산 수학의 학점은 최종 성적이 90점 이상이면 A, 80점 이상 90점 미만이며 B, 70점 이상 80점 미만이면 C, 그리고 70점 미만이면 F로 주어진다. 학점을 계산하는 program을 작성하자.

Program

```
if score >= 90:
    grade = 'A'
elif score >= 80:
    grade = 'B'
elif score >= 70:
    grade = 'C'
else:
    grade = 'F'
print('당신의 최종 성적은 ', grade, '입니다')
```

4. 여러가지 자료형

4.1 list type

list type format

```
variable_name = [element_1, element_2, ... ]
```

1 행렬 형태의 자료형이며, 각 구성 성분은 서로 다른 형인 경우도 허용된다.

```
≫ my_favorite_list = [3, 17, '수학', '판타지 소설', [등산, golf]]
```

list 내의 각 원소는 string에서의 indexing 방법과 동일하게 접근할 수 있다. +, *을 사용한 list형 자료 간의 연산과 len을 사용한 길이 계산도 string과 동일하게 적용된다.

(1) list 구성요소 확인, in / not in

Example: 12의 약수로 구성된 list를 작성하고, 입력된 숫자가 약수인지를 판별하자.

```
≫ divisor_12 = [1, 2, 3, 4, 6, 12]
≫ 1 in divisor_12              #True
≫ 5 not in divisor_12          #True
```

(2) list형 원소 변경

string형 원소에서는 index를 사용한 원소의 변경이 불가능 하였으나, list형 원소에서는 가능하다.

```
≫ divisor_12[3] = 5
```

4.2 tuple type

tuple type format

```
variable_name = (element_1, element_2, ... )
```

list형 자료에서는 각 구성 성분을 변경할 수 있다. 일부, list형에서는 원소를 변경할 수 없게 만들어야 하는 경우가 있다. 이러한 경우에는 tuple형을 사용하면 편리하다. 예를 들어, 1년을 나타내는 자료형의 경우, 고정된 원소로 구성되어 있다 (즉, 변경 불가: read-only).

```
≫ month = ('Jan', 'Feb', 'Mar', ..., 'Dec')
≫ medal = ('Gold', 'Silver', 'Bronze')
≫ single_element_tuple = (element1, )
```

위 두 예제에서, tuple 내의 각 원소의 순서도 중요한 의미를 갖는다는 것을 알 수 있다. 따라서, tuple형은 연산에서 고정된 자료로 구성된 table을 정의하는데 사용된다. 하나의 원소만을 갖는 tuple을 표현할 때는 반드시 쉼표를 같이 포함하여야 한다.

(1) tuple형의 연산

1. indexing: list의 경우와 동일
2. slice
3. + / *

Example:

```
≫ t = (1, 2, 3, 4, 5, 6)
≫ t2 = (7, )
≫ t[2]              # 3: indexing은 항상 '[ ]' 부호를 사용
≫ t[1:2]            # (2, ): tutle을 slice하면 결과 역시 tutle이 된다.
≫ t + t2            # (1, 2, 3, 4, 5, 6, 7)
≫ t2*2              # (7, 7)
≫ len(t)            # 6
```

4.3 set type

set type format

```
variable_name = {element_1, element_2, ... }
```

set형 자료에서는 기본적인 집합과 마찬가지로, 원소들 간의 순서가 없으며, 원소의 중복은 허용되지 않는다. set형 원소는 int형, float형, string형, list형, tuple형 등, 모든 자료형이 될 수 있다. 원소의 구성이 모두 같은 자료형으로 이루어져도 되고 서로 다른 자료형을 혼용하는 것도 가능하다. 또한, 원소가 하나도 없는 set형도 생성할 수 있다.

```
≫ odd_num = {1, 3, 5, 7}
≫ fruit = {'apple', 'banana', 'peach'}
≫ set_hello = set('hello')        # ('h', 'l', 'o', 'e')
≫ set_number = set([1, 2, 3, 4, 5, 4, 3])          # (1, 2, 3, 4, 5)
```

set형의 연산

1. indexing and slicing: 적용 불가
2. 원소의 추가 및 제거 가능
3. 집합 연산

Example:

```
≫ fruit = {'apple', 'banana', 'strawberry', orange'}
≫ is_banna = 'banana' in fruit
≫ fruit.remove('banana')
≫ fruit.add('melon')
≫ len(fruit)

≫ students = {'A', 'B', 'C', 'D', 'E'}
≫ len(students)
```

4.4 dictionary type

dict type format

```
variable_name = {key_1: value_1, key_2: value_2, ... }
```

1 Dictionary는 "Key-Value" 쌍을 element로 갖는 data 구조이다.

2 Dictionary는 정렬되지 않은 "Key-Value" 쌍들의 조합으로 생각할 수 있다.

Dictionary는 'dict' class로 구현된다. 일반적인 수열(sequence)에서는 배열 순서 등에 의하여 indexing 되지만, dictionary는 key 값을 기준으로 indexing 된다. 따라서, key 값은 변경할 수 없다 (immutable). 반면에, value값은 변경이 가능하다. 이러한 이유 때문에, dictionary의 key로 string(문자열)이나 tuple은 사용될 수 있는 반면, list는 사용할 수 없다.
Dictionary를 사용하면, 먼저 value와 이에 상응하는 key와 결합하여 저장한 후, 나중에 key 값을 사용하여 value를 불러오는 대부분의 연산을 처리할 수 있다.

```
≫ tel = {'Moon':5916, 'Yoo':5167, 'Kim':5917}
≫ tel['Cho'] = 5919              # append a new element on the dictionary
≫ tel['Moon']                    # 5916
≫ del tell['Kim']                # delete the 'Kim':5917 element from the dictionary
≫ tel.keys()                     # it returns a list of all keys used in the dictionary
```

Dictionary는 database를 구성하고, 관련한 작업을 수행할 때, 활용될 수 있다.

```
≫ scores = {'Kim':90, 'Yoo':98, 'Park':59}
≫ v = scores.get('Kim')          # 90
≫ if 'Kim' in scores:
...     print(scores['Kim']
...                              # 90
```

5. 반복문

5.1 while 문

while statement format

```
while condition:
        True statements
Next statements
```

1. Condition이 만족되는 동안 반복해서 'True statement'를 수행하고, 조건이 만족되지 않으면 (즉, false이면) 'Next statements'를 실행
2. while 문의 조건식이 항상 참인 경우, 'True statement'는 끝나지 않고 계속 반복 수행된다. 이러한 경우를 무한 루프라 한다.

Example: 1에서 100까지를 더하는 문제를 생각해보자. 가장 단순한 방법으로는

sum = 1 + 2 + … + 100

으로 계산할 수 있지만, 100개의 숫자와 99개의 연산자를 일일이 적기가 불편할 것임을 알 수 있다. Programming에서는 이런 문제를 변수와 조건문을 활용하여 쉽게 해결할 수 있다.

Program

```
count = 1
sum = 0
while count <= 100:
    sum = sum + count
    count = count + 1
print(sum)
```

Example: 오물렛 재료로 구성된 list에서 모든 재료를 한번씩 출력하는 program을 작성하자.

```
omlet = ['egg', 'meat', 'onion', 'carrot']
index = 0
while index <= len(omlet):
    ingredient = omlet[index]
    print(ingredient)
    index += 1
```

Example: 변수 x의 초기값이 10이고 10부터 1까지의 숫자를 출력하는 program을 작성하려 했는데 실수로 다음과 같은 program을 구성하였다.

```
x = 10
while x < 0:
    print(x)
    x = x – 1
```

5.2 for statement

for statement format

```
for item in sequence:
        True statements
```

1 'sequence' 내의 'item' 각각에 대하여 'True statement'를 반복적으로 수행한다.

for문과 while문은 서로 변환이 가능하다. for 문은 시작 값과 종료 값 등으로 사전에 지정된 범위에서 반복적으로 작업을 수행한다. 그러므로, 반복의 횟수나 범위를 미리 알고 있을 경우에는 for 문을 사용하는 것이 편리하다.

(1) range() 함수를 활용한 for 문

range statement format

```
range ([start, ]stop[, step])
```

1 'start'부터 'step' 만큼 증가/감소하면서 'stop' 전까지의 범위를 지정하는 함수
2 하나의 argument만 있을 경우는 'stop' 값을 표시한다 (나머지 두 변수는 default 값, start=0, step=1의 값을 갖는다).
3 두 개의 argument만 있을 경우는 'start'와 'stop' 값을 의미하고, 'step'값은 default 값인 1로 설정된다.

주의사항

1. range() 함수를 통한 범위 설정에서, start 값은 포함되지만 stop 값은 포함되지 않음을 주의해야 한다.
2. start/stop에는 int형 만을 사용할 수 있다.
3. start/step은 생략할 수 있다. start가 생략되는 경우 start 값은 0이라 간주한다 (default 값).

4. step값은 0이 될 수 없으며, default 값은 1이다.

```
≫ range(1, 9, 2)              # [1, 3, 5, 7]
≫ range(5, 10)                # [5, 6, 7, 8, 9]
≫ range(5)                    # [0, 1, 2, 3, 4]
≫ list(range(10))             # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] ... 총 반복 수행 수를 정의
≫ for count in range(11):
...     print(count)

≫ sum = 0
≫ for i in range(11):
...     sum += i
≫ print(sum)                  # 0 + 1 + ... + 10

≫ numbers = range(2, 11, 2)
≫ for x in numbers:
...     print(x)
```

Example: "You did a good job!"이라는 문자열을 사용자로부터 입력 받은 반복 횟수만큼 출력하는 program을 작성

```
≫ rerpeat_num = int(input('Enter the number of times to repeat: '))
≫ for i in range(repeat_num):
      print('You did a good job!')
```

(2) list를 사용한 for문

```
≫ animal = ['dog', 'cat', 'bird']
≫ for element in animal:
...     print('I love a ' + element)
≫ type(element)
```

(3) 문자열을 사용한 for 문

```
≫ for letter in 'Python':
...     print('Current letter :', letter)
```

(4) break / continue

for문 안에서 for-loop를 빠져 나오기 위하여 break 문을 사용할 수 있다. 또한 continue 문을 사용하면 loop-block의 나머지 문장들을 실행하기 않고 다음 loop로 직접 들어가게 할 수 있다.

```
≫ i = 0
≫ sum = 0
≫ while True:
...     i += 1
```

```
...     if i == 5:          # i=5이면, 다시 while문으로 이동: 수의 합산에서 5는 제외
...         continue
...     if i > 10:
...         break           # i>10이면, while문의 조건과 관계없이 바로 while문을 빠져 나온다.
...     sum += i
...
>> print(sum)              # 50
```

6. 함수, function

복잡한 program을 module(작은 program)화 하여 전체 program을 구성하는 경우, 이러한 작은 program 각각을 함수(function)라 한다.

함수의 정의와 호출

function format

```
def Function_name(n1, n2, ...)
        Statements
        return Result
```

1 함수 이름이 'Function_name'인 함수 정의

2 괄호내의 parameter(인자 or argument)인 n1, n2 등의 입력 값을 받아 그 결과 값인 'Result'를 반환하고 함수를 종결한다.

<Table 6.1> 함수 정의 시 주의사항

함수 이름, Function_Name	- 함수가 수행하는 작업을 반영하는 이름을 사용하는 것이 좋다. - 함수 이름은 변수명 작성 규칙을 따른다.
인자, parameter	- 인자는 여러 개 사용할 수 있다. - 인자를 사용하지 않을 수 있다. 이때는 인자를 생략하고 괄호만 표기한다.
return문	- 결과값을 반환하는 문장 - return문을 사용하면 함수가 종료된다. - 값을 반환하지 않고, 작업만 수행하는 함수도 있다. 이때는 return문 전체를 생략한다.

```
def average(num1, num2):
    result = (num1 + num2)/2
    return result
```

괄호안의 인자인 num1, num2를 입력 값으로 받아, 연산을 수행하고, 연산 결과인, result 값을 반환한 후 함수를 종결한다. 함수 호출은 함수명과 입력변수를 지정함으로써 이루어진다. 위 예의 함수 호출은 다음과 같다.

```
≫ mid = average(10, 20)
≫ goal = average(2, 4) * 2
```

return이 있는 함수는 결과 값으로 대체되기 때문에, 연산식에서 하나의 변수로 사용될 수도 있다.

Example: Fibonacci number 출력하는 함수를 작성

```
def fib(n)
    result = []
    a, b = 0, 1
    while a<n:
        result.append(a)
        a, b = b, a+b
    return result
```

```
≫ f2000 = fib(2000)
```

7. Modules

7.1 module 작성 및 활용

Interactive mode에서 사용한 Python 변수들은 program 종료 후, 모두 computer memory에서 삭제된다. 따라서, 긴 함수나 program을 작성하는 경우에는, text editor를 사용하여 program을 작성하고 file로 보관한 후 (통상 이러한 작업을 "script를 생성한다"고 한다), 필요할 때 마다 Python에서 불러 쓰면 편리하다.

```
# Fibonacci numbers module
def fib(n):     # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):    # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

이렇게 작성된 file을 Python에서는 module이라 칭하며, 통상 file name에 '.py' suffix를 붙인다. 예를 들어, text editor를 사용하여 'fibo.py' file을 작성하여, working directory에 저장하였다면, Python의 program이나 interactive mode에서

```
≫ import fibo
```

명령문을 사용하여 불러 들일 수 있다.

```
≫ fibo.fib(1000)      # 0 1 1 2 3 5 .... 610 987
≫ fibo.fib2(100)      # [0, 1, 1, 2, 3, 5, .... 89]
```

7.2 module search path

Programmer가 특정 module(e.g., 'fibo')을 import할 때, Python interpreter는 다음과 같은 경로를 순서대로 검색한다.

여기에서 찾을 수 없으며, 'sys.path' 변수로 설정된 directory list에서 해당 file을 찾는다. 'sys.path'는 다음 위치에서 초기화 된다.

1. Built-in module을 먼저 검색한다.
2. 입력 script를 포함하는 directory (*i.e.*, current working directory)
2. 'PYTHONPATH'(a list of directory names to search)에서 지정된 경로
4. Python이 설치된 경로 및 그 하위 library 경로

이러한 경로들은 모두 'sys.path'에 list 형태로 저장된다. 따라서, module이 검색되는 경로는 'sys.path'를 점검하면 쉽게 알 수 있다.

Current working directory는 다음 명령문을 사용하여 검색할 수 있다. 일단 os를 import하면, 일반적인 dos 명령문을 사용할 있다.

```
≫ import os
≫ os.getcwd()         # current working directory
≫ os.chdir('₩...')    # change directory
```

예를 들어, 'fib.py' script가 'D:₩Python' directory에 저장되어 있다면, 다음과 같이 'sys.path'에 경로를 추가할 수 있다.

```
≫ import sys
≫ sys.path
≫ sys.path.append('D:₩Python')
≫ import fib
```

7.3 Standard modules

Python은 표준 module library를 제공한다. 표준 module library 중에서 'math' module은 공학도들이 많이 활용할 수 있는 함수를 포함하고 있다.

```
≫ import math
≫ dir(math)                 # All math modules in the library
≫ math.cos(math.pi/4)       # 0.70710678118654757
≫ math.log(1024, 2)         # 10.0
```