**파이썬 정리**

**문자열**

**문자열 안에 작은따옴표나 큰따옴표를 포함시키고 싶을 때**

1) 문자열에 작은따옴표 (') 포함시키기

"Python's favorite food is perl"

2) 문자열에 큰따옴표 (") 포함시키기

'"Python is very easy." he says.'

3) 백슬래시(\)를 사용해서 작은따옴표(')와 큰따옴표(")를 문자열에 포함시키기

>>> food = 'Python\'s favorite food is perl'

>>> say = "\"Python is very easy.\" he says."

**여러 줄인 문자열을 변수에 대입하고 싶을 때**

1) 줄을 바꾸기 위한 이스케이프 코드 \n 삽입하기

>>> multiline = "Life is too short\nYou need python"

2) 연속된 작은따옴표 3개(''') 또는 큰따옴표 3개(""") 사용하기

작은따옴표 3개를 사용한 경우 >>> multiline='''

... Life is too short

... You need python

... '''

큰따옴표 3개를 사용한 경우 >>> multiline="""

... Life is too short

... You need python

... """

**문자열 곱하기**

>>> a = "python"

>>> a \* 2

'pythonpython'

**문자열 길이 구하기**

>>> a = "Life is too short"

>>> len(a)

17

**문자열 인덱싱과 슬라이싱**

>>> a = "Life is too short, You need Python"

>>> a[0:4] ## 0 <= a < 4 를 의미

'Life'

**문자열 포매팅**

1) 숫자 바로 대입

>>> "I eat %d apples." % 3

'I eat 3 apples.'

2) 문자열 바로 대입

>>> "I eat %s apples." % "five"

'I eat five apples.'

**※ 문자열을 대입할 때는 앞에서 배운 것처럼 큰따옴표나 작은따옴표를 반드시 써주어야 한다.**

3) 숫자 값을 나타내는 변수로 대입

>>> number = 3

>>> "I eat %d apples." % number

'I eat 3 apples.'

4) 2개 이상의 값 넣기

>>> number = 10

>>> day = "three"

>>> "I ate %d apples. so I was sick for %s days." % (number, day)

'I ate 10 apples. so I was sick for three days.'

**문자열 포맷 코드**

코드 설명

%s 문자열(String)

%c 문자 1개(character)

%d 정수(Integer)

%f 부동소수(floating-point)

%o 8진수

%x 16진수

%% Literal % (문자 % 자체)

여기에서 재미있는 것은 %s 포맷 코드인데, 이 코드는 어떤 형태의 값이든 변환해 넣을 수 있다. 무슨 말인지 예를 통해 확인해 보자.

>>> "I have %s apples" % 3

'I have 3 apples'

>>> "rate is %s" % 3.234

'rate is 3.234'

3을 문자열 안에 삽입하려면 %d를 사용하고, 3.234를 삽입하려면 %f를 사용해야 한다. 하지만 %s를 사용하면 이런 것을 생각하지 않아도 된다. 왜냐하면 %s는 자동으로 % 뒤에 있는 값을 문자열로 바꾸기 때문이다.

**format 함수를 사용한 포매팅**

\* 인덱스와 이름을 혼용해서 넣기

>>> "I ate {0} apples. so I was sick for {day} days.".format(10, day=3)

'I ate 10 apples. so I was sick for 3 days.'

위와 같이 인덱스 항목과 name=value 형태를 혼용하는 것도 가능하다.

\* 왼쪽 정렬

>>> "{0:<10}".format("hi")

'hi '

\* 오른쪽 정렬

>>> "{0:>10}".format("hi")

' hi'

\* 가운데 정렬

>>> "{0:^10}".format("hi")

' hi '

\*공백 채우기

>>> "{0:=^10}".format("hi")

'====hi===='

>>> "{0:!<10}".format("hi")

'hi!!!!!!!!'

\*소수점 표현하기

>>> y = 3.42134234

>>> "{0:0.4f}".format(y)

'3.4213'

위 예는 format 함수를 사용해 소수점을 4자리까지만 표현하는 방법을 보여 준다. 앞에서 살펴보았던 표현식 0.4f를 그대로 사용한 것을 알 수 있다.

>>> "{0:10.4f}".format(y)

' 3.4213'

위와 같이 자릿수를 10으로 맞출 수도 있다. 역시 앞에서 살펴본 10.4f의 표현식을 그대로 사용한 것을 알 수 있다.

\* { 또는 } 문자 표현하기

>>> "{{ and }}".format()

'{ and }'

format 함수를 사용해 문자열 포매팅을 할 경우 { }와 같은 중괄호(brace) 문자를 포매팅 문자가 아닌 문자 그대로 사용하고 싶은 경우에는 위 예의 {{ }}처럼 2개를 연속해서 사용하면 된다.

**문자열 관련 함수들**

\* 문자 개수 세기(count)

>>> a = "hobby"

>>> a.count('b')

2

\* 위치 알려주기1(find)

>>> a = "Python is the best choice"

>>> a.find('b')

14

>>> a.find('k')

-1

\* 위치 알려주기2(index)

>>> a = "Life is too short"

>>> a.index('t')

8

>>> a.index('k')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: substring not found

\* 문자열 삽입(join)

>>> ",".join('abcd')

'a,b,c,d'

>>> ",".join(['a', 'b', 'c', 'd'])

'a,b,c,d'

\* 왼쪽 공백 지우기(lstrip)

>>> a = " hi "

>>> a.lstrip()

'hi '

\* 오른쪽 공백 지우기(rstrip)

>>> a= " hi "

>>> a.rstrip()

' hi'

\* 양쪽 공백 지우기(strip)

>>> a = " hi "

>>> a.strip()

'hi'

문자열 양쪽에 있는 한 칸 이상의 연속된 공백을 모두 지운다.

\* 문자열 바꾸기(replace)

>>> a = "Life is too short"

>>> a.replace("Life", "Your leg")

'Your leg is too short'

\* 문자열 나누기(split)

>>> a = "Life is too short"

>>> a.split()

['Life', 'is', 'too', 'short']

>>> b = "a:b:c:d"

>>> b.split(':')

['a', 'b', 'c', 'd']

**리스트**

**리스트의 수정과 삭제**

\* 리스트에서 값 수정하기

>>> a = [1, 2, 3]

>>> a[2] = 4

>>> a

[1, 2, 4]

\* del 함수 사용해 리스트 요소 삭제하기

>>> a = [1, 2, 3]

>>> del a[1]

>>> a

[1, 3]

\* del 객체

※ 객체란 파이썬에서 사용되는 모든 자료형을 말한다.

>>> a = [1, 2, 3, 4, 5]

>>> del a[2:]

>>> a

[1, 2]

a[2:]에 해당하는 리스트의 요소들이 삭제되었다.

**리스트 관련 함수들**

\* 리스트에 요소 추가(append)

>>> a = [1, 2, 3]

>>> a.append(4)

>>> a

[1, 2, 3, 4]

>>> a.append([5,6])

>>> a

[1, 2, 3, 4, [5, 6]]

\*리스트 정렬(sort)

>>> a = [1, 4, 3, 2]

>>> a.sort()

>>> a

[1, 2, 3, 4]

>>> a = ['a', 'c', 'b']

>>> a.sort()

>>> a

['a', 'b', 'c']

\*리스트 뒤집기(reverse)

>>> a = ['a', 'c', 'b']

>>> a.reverse()

>>> a

['b', 'c', 'a']

\* 위치 반환(index)

>>> a = [1,2,3]

>>> a.index(3)

2

>>> a.index(0)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: 0 is not in list

\*리스트에 요소 삽입(insert)

>>> a = [1, 2, 3]

>>> a.insert(0, 4)

>>> a

[4, 1, 2, 3]

>>> a.insert(3, 5)

>>> a

[4, 1, 2, 5, 3]

\* 리스트 요소 제거(remove)

remove(x)는 리스트에서 첫 번째로 나오는 x를 삭제하는 함수이다.

>>> a = [1, 2, 3, 1, 2, 3]

>>> a.remove(3)

>>> a

[1, 2, 1, 2, 3]

\* 리스트 요소 끄집어내기(pop)

>>> a = [1,2,3]

>>> a.pop()

3

>>> a

[1, 2]

\* 리스트에 포함된 요소 x의 개수 세기(count)

>>> a = [1,2,3,1]

>>> a.count(1)

2

\* 리스트 확장(extend)

>>> a = [1,2,3]

>>> a.extend([4,5])

>>> a

[1, 2, 3, 4, 5]

>>> b = [6, 7]

>>> a.extend(b)

>>> a

[1, 2, 3, 4, 5, 6, 7]

a.extend([4, 5])는 a += [4, 5]와 동일하다.

**튜플**

튜플(tuple)은 몇 가지 점을 제외하곤 리스트와 거의 비슷하며 리스트와 다른 점은 다음과 같다.

* 리스트는 [ ]으로 둘러싸지만 튜플은 ( )으로 둘러싼다.
* 리스트는 그 값의 생성, 삭제, 수정이 가능하지만 튜플은 그 값을 바꿀 수 없다.

>>> t1 = ()

>>> t2 = (1,)

>>> t3 = (1, 2, 3)

>>> t4 = 1, 2, 3

>>> t5 = ('a', 'b', ('ab', 'cd'))

* t2 = (1,)처럼 단지 1개의 요소만을 가질 때는 요소 뒤에 콤마(,)를 반드시 붙여야 한다는 것
* t4 = 1, 2, 3처럼 괄호( )를 생략해도 무방

**딕셔너리**

{Key1:Value1, Key2:Value2, Key3:Value3, ...}

Key와 Value의 쌍 여러 개가 { }로 둘러싸여 있다. 각각의 요소는 Key : Value 형태로 이루어져 있고 쉼표(,)로 구분되어 있다.

※ Key에는 변하지 않는 값을 사용하고, Value에는 변하는 값과 변하지 않는 값 모두 사용할 수 있다.

다음 딕셔너리 예를 살펴보자.

>>> dic = {'name':'pey', 'phone':'0119993323', 'birth': '1118'}

**딕셔너리 쌍 추가, 삭제하기**

\* 딕셔너리 쌍 추가하기

>>> a = {1: 'a'}

>>> a[2] = 'b'

>>> a

{1: 'a', 2: 'b'}

>>> a['name'] = 'pey'

>>> a

{1: 'a', 2: 'b', 'name': 'pey'}

\* 딕셔너리 요소 삭제하기

>>> del a[1]

>>> a

{2: 'b', 'name': 'pey', 3: [1, 2, 3]}

**딕셔너리에서 Key 사용해 Value 얻기**

>>> dic = {'name':'pey', 'phone':'0119993323', 'birth': '1118'}

>>> dic['name']

'pey'

>>> dic['phone']

'0119993323'

>>> dic['birth']

'1118'

**딕셔너리 만들 때 주의할 사항**

* 중복되는 Key 값을 설정해 놓으면 하나를 제외한 나머지 것들이 모두 무시된다는 점을 주의해야 한다. 다음 예에서 볼 수 있듯이 동일한 Key가 2개 존재할 경우 1:'a' 쌍이 무시된다.

>>> a = {1:'a', 1:'b'}

>>> a

{1: 'b'}

* Key에 리스트는 쓸 수 없다는 것이다. 하지만 튜플은 Key로 쓸 수 있다. 딕셔너리의 Key로 쓸 수 있느냐 없느냐는 Key가 변하는 값인지 변하지 않는 값인지에 달려 있다. 리스트는 그 값이 변할 수 있기 때문에 Key로 쓸 수 없다.

>>> a = {[1,2] : 'hi'}

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: unhashable type: 'list'

**딕셔너리 관련 함수들**

\* Key 리스트 만들기(keys)

>>> a = {'name': 'pey', 'phone': '0119993323', 'birth': '1118'}

>>> a.keys()

dict\_keys(['name', 'phone', 'birth'])

\* dict\_keys 객체를 리스트로 변환하려면 다음과 같이 하면 된다.

>>> list(a.keys())

['name', 'phone', 'birth']

Value 리스트 만들기(values)

>>> a.values()

dict\_values(['pey', '0119993323', '1118'])

\* Key, Value 쌍 얻기(items)

>>> a.items()

dict\_items([('name', 'pey'), ('phone', '0119993323'), ('birth', '1118')])

\* Key: Value 쌍 모두 지우기(clear)

>>> a.clear()

>>> a

{}

clear 함수는 딕셔너리 안의 모든 요소를 삭제한다. 빈 리스트를 [ ], 빈 튜플을 ( )로 표현하는 것과 마찬가지로 빈 딕셔너리도 { }로 표현한다.

\* Key로 Value얻기(get)

>>> a = {'name':'pey', 'phone':'0119993323', 'birth': '1118'}

>>> a.get('name')

'pey'

>>> a.get('phone')

'0119993323'

get(x) 함수는 x라는 Key에 대응되는 Value를 돌려준다. 앞에서 살펴보았듯이 a.get('name')은 a['name']을 사용했을 때와 동일한 결괏값을 돌려받는다.

다만 다음 예제에서 볼 수 있듯이 a['nokey']처럼 존재하지 않는 키(nokey)로 값을 가져오려고 할 경우 a['nokey']는 Key 오류를 발생시키고 a.get('nokey')는 None을 돌려준다는 차이가 있다. 어떤것을 사용할지는 여러분의 선택이다.

※ 여기에서 None은 "거짓"이라는 뜻이라고만 알아두자.

>>> a = {'name':'pey', 'phone':'0119993323', 'birth': '1118'}

>>> print(a.get('nokey'))

None

>>> print(a['nokey'])

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

KeyError: 'nokey'

딕셔너리 안에 찾으려는 Key 값이 없을 경우 미리 정해 둔 디폴트 값을 대신 가져오게 하고 싶을 때에는 get(x, '디폴트 값')을 사용하면 편리하다.

>>> a.get('foo', 'bar')

'bar'

a 딕셔너리에는 'foo'에 해당하는 값이 없다. 따라서 디폴트 값인 'bar'를 돌려준다.

\* 해당 Key가 딕셔너리 안에 있는지 조사하기(in)

>>> a = {'name':'pey', 'phone':'0119993323', 'birth': '1118'}

>>> 'name' in a

True

>>> 'email' in a

False

**집합**

집합 자료형은 다음과 같이 set 키워드를 사용해 만들 수 있다.

>>> s1 = set([1,2,3])

>>> s1

{1, 2, 3}

>>> s2 = set("Hello")

>>> s2

{'e', 'H', 'l', 'o'}

※ 비어 있는 집합 자료형은 s = set()로 만들수 있다.

* 중복을 허용하지 않는다.
* 순서가 없다(Unordered).

**교집합, 합집합, 차집합 구하기**

>>> s1 = set([1, 2, 3, 4, 5, 6])

>>> s2 = set([4, 5, 6, 7, 8, 9])

1) 교집합

>>> s1 & s2 >>> s1.intersection(s2)

{4, 5, 6}

2) 합집합

>>> s1 | s2 >>> s1.union(s2)

{1, 2, 3, 4, 5, 6, 7, 8, 9}

3) 차집합

>>> s1 - s2 >>> s1.difference(s2)

{1, 2, 3}

**집합 자료형 관련 함수들**

\* 값 1개 추가하기(add)

>>> s1 = set([1, 2, 3])

>>> s1.add(4)

>>> s1

{1, 2, 3, 4}

\* 값 여러 개 추가하기(update)

>>> s1 = set([1, 2, 3])

>>> s1.update([4, 5, 6])

>>> s1

{1, 2, 3, 4, 5, 6}

\* 특정 값 제거하기(remove)

>>> s1 = set([1, 2, 3])

>>> s1.remove(2)

>>> s1

{1, 3}

**불**

**자료형의 참과 거짓**

값 참 or 거짓

"python" 참

"" 거짓

[1, 2, 3] 참

[] 거짓

() 거짓

{} 거짓

1 참

0 거짓

None 거짓

**변수**

**리스트를 복사하고자 할 때**

>>> a = [1,2,3]

>>> b = a

b는 a와 완전히 동일하다고 할 수 있다. 다만 [1, 2, 3] 리스트를 참조하는 변수가 a 변수 1개에서 b 변수가 추가되어 2개로 늘어났다는 차이만 있을 뿐이다.

id 함수를 사용하면 이러한 사실을 확인할 수 있다.

>>> id(a)

4303029896

>>> id(b)

4303029896

**if 문**

**x in s, x not in s**

>>> 1 in [1, 2, 3]

True

>>> 1 not in [1, 2, 3]

False

**for 문**

**다양한 for문의 사용**

>>> a = [(1,2), (3,4), (5,6)]

>>> for (first, last) in a:

... print(first + last)

...

3

7

11

**for문과 함께 자주 사용하는 range 함수**

>>> a = range(10)

>>> a

range(0, 10)

range(10)은 0부터 10 미만의 숫자를 포함하는 range 객체를 만들어 준다.

**리스트 내포 사용하기**

>>> a = [1,2,3,4]

>>> result = [num \* 3 for num in a if num % 2 == 0]

>>> print(result)

[6, 12]

>>> result = [x\*y for x in range(2,10)

... for y in range(1,10)]

>>> print(result)

[2, 4, 6, 8, 10, 12, 14, 16, 18, 3, 6, 9, 12, 15, 18, 21, 24, 27, 4, 8, 12, 16,

20, 24, 28, 32, 36, 5, 10, 15, 20, 25, 30, 35, 40, 45, 6, 12, 18, 24, 30, 36, 42

, 48, 54, 7, 14, 21, 28, 35, 42, 49, 56, 63, 8, 16, 24, 32, 40, 48, 56, 64, 72,

9, 18, 27, 36, 45, 54, 63, 72, 81]

**함수**

**여러 개의 입력값을 받는 함수 만들기**

>>> def add\_many(\*args):

... result = 0

... for i in args:

... result = result + i

... return result

...

>>>

**키워드 파라미터 kwargs**

>>> def print\_kwargs(\*\*kwargs):

... print(kwargs)

...

>>> print\_kwargs(a=1)

{'a': 1}

>>> print\_kwargs(name='foo', age=3)

{'age': 3, 'name': 'foo'}

입력값 a=1 또는 name='foo', age=3이 모두 딕셔너리로 만들어져서 출력된다는 것을 확인할 수 있다. 즉 \*\*kwargs처럼 매개변수 이름 앞에 \*\*을 붙이면 매개변수 kwargs는 딕셔너리가 되고 모든 key=value 형태의 결괏값이 그 딕셔너리에 저장된다.

※ 여기에서 kwargs는 keyword arguments의 약자이며 args와 마찬가지로 관례적으로 사용한다.

**lambda**

* lambda 매개변수1, 매개변수2, ... : 매개변수를 이용한 표현식

>>> add = lambda a, b: a+b

>>> result = add(3, 4)

>>> print(result)

7

**사용자 입력과 출력**

**input의 사용**

>>> a = input()

Life is too short, you need python

>>> a

'Life is too short, you need python'

>>>

**print 자세히 알기**

* 큰따옴표(")로 둘러싸인 문자열은 + 연산과 동일하다

>>> print("life" "is" "too short") # ①

lifeistoo short

>>> print("life"+"is"+"too short") # ②

lifeistoo short

* 문자열 띄어쓰기는 콤마로 한다

>>> print("life", "is", "too short")

life is too short

**파일 읽고 쓰기**

파일열기모드 설명

r 읽기모드 - 파일을 읽기만 할 때 사용

w 쓰기모드 - 파일에 내용을 쓸 때 사용

a 추가모드 - 파일의 마지막에 새로운 내용을 추가 시킬 때 사용

* 파일을 쓰기 모드로 열면 해당 파일이 이미 존재할 경우 원래 있던 내용이 모두 사라지고, 해당 파일이 존재하지 않으면 새로운 파일이 생성된다.
* 만약 새파일.txt 파일을 C:/doit 디렉터리에 생성하고 싶다면 다음과 같이 작성해야 한다.

f = open("C:/doit/새파일.txt", 'w')

f.close()

* 위 예에서 f.close()는 열려 있는 파일 객체를 닫아 주는 역할을 한다. 사실 이 문장은 생략해도 된다. 프로그램을 종료할 때 파이썬 프로그램이 열려 있는 파일의 객체를 자동으로 닫아주기 때문이다. 하지만 close()를 사용해서 열려 있는 파일을 직접 닫아 주는 것이 좋다. 쓰기모드로 열었던 파일을 닫지 않고 다시 사용하려고 하면 오류가 발생하기 때문이다.

**with문이 파일을 열고 닫는 것을 자동으로 처리**

with open("foo.txt", "w") as f:

f.write("Life is too short, you need python")

**클래스**

class Calculator:

def \_\_init\_\_(self):

self.result = 0

def add(self, num):

self.result += num

return self.result

cal1 = Calculator()

cal2 = Calculator()

print(cal1.add(3))

print(cal1.add(4))

print(cal2.add(3))

print(cal2.add(7))

프로그램을 실행하면 함수 2개를 사용했을 때와 동일한 결과가 출력된다.

3

7

3

10

**[객체와 인스턴스의 차이]**

>>> class Cookie:

>>> pass

>>> a = Cookie()

클래스로 만든 객체를 인스턴스라고도 한다. 그렇다면 객체와 인스턴스의 차이는 무엇일까? 이렇게 생각해 보자. a = Cookie() 이렇게 만든 a는 객체이다. 그리고 a 객체는 Cookie의 인스턴스이다. 즉 인스턴스라는 말은 특정 객체(a)가 어떤 클래스(Cookie)의 객체인지를 관계 위주로 설명할 때 사용한다. "a는 인스턴스"보다는 "a는 객체"라는 표현이 어울리며 "a는 Cookie의 객체"보다는 "a는 Cookie의 인스턴스"라는 표현이 훨씬 잘 어울린다.

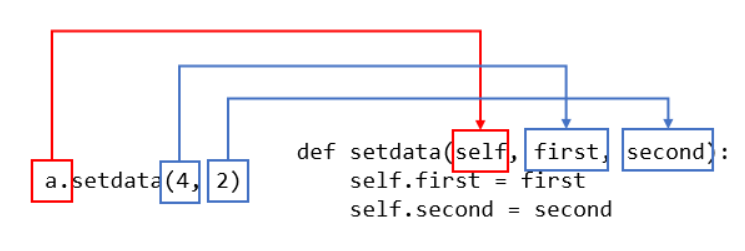
>>> class FourCal:

... def setdata(self, first, second):

... self.first = first

... self.second = second

...

>>>

**생성자 (Constructor)**

파이썬 메서드 이름으로 \_\_init\_\_를 사용하면 이 메서드는 생성자가 된다. 다음과 같이 FourCal 클래스에 생성자를 추가해 보자.

>>> class FourCal:

... def \_\_init\_\_(self, first, second):

... self.first = first

... self.second = second

... def setdata(self, first, second):

... self.first = first

... self.second = second

... def add(self):

... result = self.first + self.second

... return result

... def mul(self):

... result = self.first \* self.second

... return result

... def sub(self):

... result = self.first - self.second

... return result

... def div(self):

... result = self.first / self.second

... return result

...

>>>

\_\_init\_\_ 메서드는 setdata 메서드와 이름만 다르고 모든 게 동일하다. 단 메서드 이름을 \_\_init\_\_으로 했기 때문에 생성자로 인식되어 객체가 생성되는 시점에 자동으로 호출되는 차이가 있다.

**클래스의 상속**

어떤 클래스를 만들 때 다른 클래스의 기능을 물려받을 수 있게 만드는 것이다. 이번에는 상속 개념을 사용하여 우리가 만든 FourCal 클래스에 ab (a의 b제곱)을 구할 수 있는 기능을 추가해 보자.

class MoreFourCal(FourCal):

... def pow(self):

... result = self.first \*\* self.second

... return result

...

>>>

상속은 MoreFourCal 클래스처럼 기존 클래스(FourCal)는 그대로 놔둔 채 클래스의 기능을 확장시킬 때 주로 사용한다.

**메서드 오버라이딩**

부모 클래스(상속한 클래스)에 있는 메서드를 동일한 이름으로 다시 만드는 것을 메서드 오버라이딩(Overriding, 덮어쓰기)이라고 한다. 이렇게 메서드를 오버라이딩하면 부모클래스의 메서드 대신 오버라이딩한 메서드가 호출된다.

**클래스 변수**

클래스 변수는 클래스로 만든 모든 객체에 공유된다는 특징이 있다. 즉, 모두 같은 메모리 값을 가리킨다.

**모듈**

**if \_\_name\_\_ == "\_\_main\_\_": 의 의미**

# mod1.py

def add(a, b):

return a+b

def sub(a, b):

return a-b

if \_\_name\_\_ == "\_\_main\_\_":

print(add(1, 4))

print(sub(4, 2))

if \_\_name\_\_ == "\_\_main\_\_"을 사용하면 C:\doit>python mod1.py처럼 직접 이 파일을 실행했을 때는 \_\_name\_\_ == "\_\_main\_\_"이 참이 되어 if문 다음 문장이 수행된다. 반대로 대화형 인터프리터나 다른 파일에서 이 모듈을 불러서 사용할 때는 \_\_name\_\_ == "\_\_main\_\_"이 거짓이 되어 if문 다음 문장이 수행되지 않는다.

**\_\_name\_\_ 변수**

파이썬의 \_\_name\_\_ 변수는 파이썬이 내부적으로 사용하는 특별한 변수 이름이다. 만약 C:\doit>python mod1.py처럼 직접 mod1.py 파일을 실행할 경우 mod1.py의 \_\_name\_\_ 변수에는 \_\_main\_\_ 값이 저장된다. 하지만 파이썬 셸이나 다른 파이썬 모듈에서 mod1을 import 할 경우에는 mod1.py의 \_\_name\_\_ 변수에는 mod1.py의 모듈 이름 값 mod1이 저장된다.

**패키지**

패키지(Packages)는 도트(.)를 사용하여 파이썬 모듈을 계층적(디렉터리 구조)으로 관리할 수 있게 해준다. 예를 들어 모듈 이름이 A.B인 경우에 A는 패키지 이름이 되고 B는 A 패키지의 B모듈이 된다.

**가상의 game 패키지 예)**

game/

\_\_init\_\_.py

sound/

\_\_init\_\_.py

echo.py

wav.py

graphic/

\_\_init\_\_.py

screen.py

render.py

play/

\_\_init\_\_.py

run.py

test.py

game, sound, graphic, play는 디렉터리 이름이고 확장자가 .py인 파일은 파이썬 모듈이다. game 디렉터리가 이 패키지의 루트 디렉터리이고 sound, graphic, play는 서브 디렉터리이다.

>>> import game.sound.echo

**\_\_init\_\_.py 의 용도**

\_\_init\_\_.py 파일은 해당 디렉터리가 패키지의 일부임을 알려주는 역할을 한다. 만약 game, sound, graphic 등 패키지에 포함된 디렉터리에 \_\_init\_\_.py 파일이 없다면 패키지로 인식되지 않는다.

**예외 처리**

**오류 예외 처리 기법**

**try, except문**

try:

...

except [발생 오류[as 오류 메시지 변수]]:

...

try 블록 수행 중 오류가 발생하면 except 블록이 수행된다. 하지만 try 블록에서 오류가 발생하지 않는다면 except 블록은 수행되지 않는다. 위 구문을 보면 [ ] 기호를 사용하는데, 이 기호는 괄호 안의 내용을 생략할 수 있다는 관례 표기법이다.

1) try, except만 쓰는 방법 : 오류 종류에 상관없이 오류가 발생하면 except 블록을 수행한다.

try:

...

except:

...

2) 발생 오류만 포함한 except문 : except문에 미리 정해 놓은 오류 이름과 일치할 때만 except 블록을 수행한다는 뜻이다.

try:

...

except 발생 오류:

...

3) 발생 오류와 오류 메시지 변수까지 포함한 except문 : 두 번째 경우에서 오류 메시지의 내용까지 알고 싶을 때 사용하는 방법이다.

try:

except 발생 오류 as 오류 메시지 변수:

\* try .. finally

try문에는 finally절을 사용할 수 있다. finally절은 try문 수행 도중 예외 발생 여부에 상관없이 항상 수행된다. 보통 finally절은 사용한 리소스를 close해야 할 때에 많이 사용한다.

f = open('foo.txt', 'w')

try:

# 무언가를 수행한다.

finally:

f.close()

예외 발생 여부와 상관없이 finally절에서 f.close()로 열린 파일을 닫을 수 있다.

\* 여러개의 오류처리하기

try:

a = [1,2]

print(a[3])

4/0

except ZeroDivisionError as e:

print(e)

except IndexError as e:

print(e)

프로그램을 실행하면 "list index out of range" 오류 메시지가 출력될 것이다.

다음과 같이 ZerroDivisionError와 IndexError를 함께 처리할 수도 있다.

try:

a = [1,2]

print(a[3])

4/0

except (ZeroDivisionError, IndexError) as e:

print(e)

**오류 회피하기**

try:

f = open("나없는파일", 'r')

except FileNotFoundError:

pass

try문 안에서 FileNotFoundError가 발생할 경우에 pass를 사용하여 오류를 그냥 회피하도록 작성한 예제이다.

**오류 일부러 발생시키기**

이상하게 들리겠지만 프로그래밍을 하다 보면 종종 오류를 일부러 발생시켜야 할 경우도 생긴다. 파이썬은 raise 명령어를 사용해 오류를 강제로 발생시킬 수 있다.

예를 들어 Bird 클래스를 상속받는 자식 클래스는 반드시 fly라는 함수를 구현하도록 만들고 싶은 경우(강제로 그렇게 하고 싶은 경우)가 있을 수 있다. 다음 예를 보자.

class Bird:

def fly(self):

raise NotImplementedError

위 예제는 Bird 클래스를 상속받는 자식 클래스는 반드시 fly 함수를 구현해야 한다는 의지를 보여 준다. 만약 자식 클래스가 fly 함수를 구현하지 않은 상태로 fly 함수를 호출한다면 어떻게 될까?

※ NotImplementedError는 파이썬 내장 오류로, 꼭 작성해야 하는 부분이 구현되지 않았을 경우 일부러 오류를 일으키기 위해 사용한다.

class Eagle(Bird):

pass

eagle = Eagle()

eagle.fly()

Eagle 클래스는 Bird 클래스를 상속받는다. 그런데 Eagle 클래스에서 fly 함수를 구현하지 않았기 때문에 Bird 클래스의 fly 함수가 호출된다. 그리고 raise문에 의해 NotImplemented Error가 발생할 것이다.

※ 상속받는 클래스에서 함수를 재구현하는 것을 메서드 오버라이딩이라고 부른다.

Traceback (most recent call last):

File "...", line 33, in <module>

eagle.fly()

File "...", line 26, in fly

raise NotImplementedError

NotImplementedError

**예외 만들기**

프로그램 수행 도중 특수한 경우에만 예외 처리를 하기 위해서 종종 예외를 만들어서 사용한다. 직접 예외를 만들어 보자. 예외는 다음과 같이 파이썬 내장 클래스인 Exception 클래스를 상속하여 만들 수 있다.

class MyError(Exception):

pass

그리고 별명을 출력해 주는 함수를 다음과 같이 작성한다.

def say\_nick(nick):

if nick == '바보':

raise MyError()

print(nick)

그리고 다음과 같이 say\_nick 함수를 호출해 보자.

say\_nick("천사")

say\_nick("바보")

천사

Traceback (most recent call last):

File "...", line 11, in <module>

say\_nick("바보")

File "...", line 7, in say\_nick

raise MyError()

\_\_main\_\_.MyError

이번에는 예외 처리 기법을 사용하여 MyError 발생을 예외 처리해 보자.

try:

say\_nick("천사")

say\_nick("바보")

except MyError:

print("허용되지 않는 별명입니다.")

프로그램을 실행하면 다음과 같이 출력된다.

천사

허용되지 않는 별명입니다.

만약 오류 메시지를 사용하고 싶다면 다음처럼 예외 처리를 하면 된다.

try:

say\_nick("천사")

say\_nick("바보")

except MyError as e:

print(e)

하지만 프로그램을 실행해 보면 print(e)로 오류 메시지가 출력되지 않는 것을 확인할 수 있다. 오류 메시지를 출력했을 때 오류 메시지가 보이게 하려면 오류 클래스에 다음과 같은 \_\_str\_\_ 메서드를 구현해야 한다. \_\_str\_\_ 메서드는 print(e)처럼 오류 메시지를 print문으로 출력할 경우에 호출되는 메서드이다.

class MyError(Exception):

def \_\_str\_\_(self):

return "허용되지 않는 별명입니다."

프로그램을 다시 실행해 보면 "허용되지 않는 별명입니다."라는 오류메시지가 출력되는 것을 확인할 수 있다.

**내장함수**

**all**

all(x)는 반복 가능한(iterable) 자료형 x를 입력 인수로 받으며 이 x가 모두 참이면 True, 거짓이 하나라도 있으면 False를 돌려준다.

※ 반복 가능한 자료형이란 for문으로 그 값을 출력할 수 있는 것을 의미한다. 리스트, 튜플, 문자열, 딕셔너리, 집합 등이 있다.

>>> all([1, 2, 3])

True

리스트 자료형 [1, 2, 3]은 모든 요소가 참이므로 True를 돌려준다.

>>> all([1, 2, 3, 0])

False

리스트 자료형 [1, 2, 3, 0] 중에서 요소 0은 거짓이므로 False를 돌려준다.

**any**

all(x)의 반대이다.

>>> any([1, 2, 3, 0])

True

>>> any([0, ""])

False

리스트 자료형 [0, ""]의 요소 0과 ""은 모두 거짓이므로 False를 돌려준다.

**chr**

chr(i)는 아스키(ASCII) 코드 값을 입력받아 그 코드에 해당하는 문자를 출력하는 함수이다.

>>> chr(97)

'a'

**dir**

dir은 객체가 자체적으로 가지고 있는 변수나 함수를 보여 준다. 다음 예는 리스트와 딕셔너리 객체 관련 함수(메서드)를 보여 주는 예이다. 우리가 02장에서 살펴본 자료형 관련 함수를 만나 볼 수 있다.

>>> dir([1, 2, 3])

['append', 'count', 'extend', 'index', 'insert', 'pop',...]

>>> dir({'1':'a'})

['clear', 'copy', 'get', 'has\_key', 'items', 'keys',...]

**divmod**

divmod(a, b)는 2개의 숫자를 입력으로 받는다. 그리고 a를 b로 나눈 몫과 나머지를 튜플 형태로 돌려주는 함수이다.

>>> divmod(7, 3)

(2, 1)

**enumerate**

enumerate는 "열거하다"라는 뜻이다. 이 함수는 순서가 있는 자료형(리스트, 튜플, 문자열)을 입력으로 받아 인덱스 값을 포함하는 enumerate 객체를 돌려준다.

>>> for i, name in enumerate(['body', 'foo', 'bar']):

... print(i, name)

...

0 body

1 foo

2 bar

순서 값과 함께 body, foo, bar가 순서대로 출력되었다. 즉 위 예제와 같이 enumerate를 for문과 함께 사용하면 자료형의 현재 순서(index)와 그 값을 쉽게 알 수 있다.

**eval**

eval(expression )은 실행 가능한 문자열(1+2, 'hi' + 'a' 같은 것)을 입력으로 받아 문자열을 실행한 결괏값을 돌려주는 함수이다.

>>> eval('1+2')

3

보통 eval은 입력받은 문자열로 파이썬 함수나 클래스를 동적으로 실행하고 싶을 때 사용한다.

**filter**

filter 함수는 첫 번째 인수로 함수 이름을, 두 번째 인수로 그 함수에 차례로 들어갈 반복 가능한 자료형을 받는다. 그리고 두 번째 인수인 반복 가능한 자료형 요소가 첫 번째 인수인 함수에 입력되었을 때 반환 값이 참인 것만 묶어서(걸러 내서) 돌려준다.

#filter1.py

def positive(x):

return x > 0

print(list(filter(positive, [1, -3, 2, 0, -5, 6])))

결과값: [1, 2, 6]

여기에서는 두 번째 인수인 리스트의 요소들이 첫 번째 인수인 positive 함수에 입력되었을때 반환 값이 참인 것만 묶어서 돌려준다. 앞의 예에서는 1, 2, 6만 양수여서 x > 0 문장이 참이되므로 [1, 2, 6]이라는 결괏값을 돌려주게 된 것이다.

**int**

int(x)는 문자열 형태의 숫자나 소수점이 있는 숫자 등을 정수 형태로 돌려주는 함수로, 정수를 입력으로 받으면 그대로 돌려준다.

>>> int('3')

3

>>> int(3.4)

3

int(x, radix)는 radix 진수로 표현된 문자열 x를 10진수로 변환하여 돌려준다.

2진수로 표현된 11의 10진수 값은 다음과 같이 구한다.

>>> int('11', 2)

3

16진수로 표현된 1A의 10진수 값은 다음과 같이 구한다.

>>> int('1A', 16)

26

**isinstance**

isinstance(object, class )는 첫 번째 인수로 인스턴스, 두 번째 인수로 클래스 이름을 받는다. 입력으로 받은 인스턴스가 그 클래스의 인스턴스인지를 판단하여 참이면 True, 거짓이면 False를 돌려준다.

>>> class Person: pass

...

>>> a = Person()

>>> isinstance(a, Person)

True

위 예는 a가 Person 클래스가 만든 인스턴스임을 확인시켜 준다.

>>> b = 3

>>> isinstance(b, Person)

False

b는 Person 클래스가 만든 인스턴스가 아니므로 False를 돌려준다.

**list**

list(s)는 반복 가능한 자료형 s를 입력받아 리스트로 만들어 돌려주는 함수이다.

>>> list("python")

['p', 'y', 't', 'h', 'o', 'n']

list 함수에 리스트를 입력으로 주면 똑같은 리스트를 복사하여 돌려준다.

>>> a = [1, 2, 3]

>>> b = list(a)

>>> b

[1, 2, 3]

**map**

map(f, iterable)은 함수(f)와 반복 가능한(iterable) 자료형을 입력으로 받는다. map은 입력받은 자료형의 각 요소를 함수 f가 수행한 결과를 묶어서 돌려주는 함수이다.

# two\_times.py

def two\_times(numberList):

result = [ ]

for number in numberList:

result.append(number\*2)

return result

result = two\_times([1, 2, 3, 4])

print(result)

two\_times 함수는 리스트 요소를 입력받아 각 요소에 2를 곱한 결괏값을 돌려준다. 실행 결과는 다음과 같다.

결과값: [2, 4, 6, 8]

위 예제는 map 함수를 사용하면 다음처럼 바꿀 수 있다.

>>> def two\_times(x):

... return x\*2

...

>>> list(map(two\_times, [1, 2, 3, 4]))

[2, 4, 6, 8]

이제 앞 예제를 해석해 보자. 먼저 리스트의 첫 번째 요소인 1이 two\_times 함수의 입력값으로 들어가고 1 \* 2의 과정을 거쳐서 2가 된다. 다음으로 리스트의 두 번째 요소인 2가 2 \* 2 의 과정을 거쳐 4가 된다. 따라서 결괏값 리스트는 이제 [2, 4]가 된다. 총 4개의 요솟값이 모두 수행되면 마지막으로 [2, 4, 6, 8]을 돌려준다. 이것이 map 함수가 하는 일이다.

>>> list(map(lambda a: a\*2, [1, 2, 3, 4]))

**max**

max(iterable)는 인수로 반복 가능한 자료형을 입력받아 그 최댓값을 돌려주는 함수이다.

>>> max([1, 2, 3])

3

**min**

min(iterable)은 max 함수와 반대로, 인수로 반복 가능한 자료형을 입력받아 그 최솟값을 돌려주는 함수이다.

>>> min([1, 2, 3])

1

**open**

open(filename, [mode])은 "파일 이름"과 "읽기 방법"을 입력받아 파일 객체를 돌려주는 함수이다. 읽기 방법(mode)을 생략하면 기본값인 읽기 전용 모드(r)로 파일 객체를 만들어 돌려준다.

mode 설명

w 쓰기 모드로 파일 열기

r 읽기 모드로 파일 열기

a 추가 모드로 파일 열기

b 바이너리 모드로 파일 열기

b는 w, r, a와 함께 사용한다.

>>> f = open("binary\_file", "rb")

위 예의 rb는 "바이너리 읽기 모드"를 의미한다.

**ord**

ord(c)는 문자의 아스키 코드 값을 돌려주는 함수이다.

※ ord 함수는 chr 함수와 반대이다.

>>> ord('a')

97

**range**

인수가 3개일 경우

세 번째 인수는 숫자 사이의 거리를 말한다.

>>> list(range(1, 10, 2))

[1, 3, 5, 7, 9]

>>> list(range(0, -10, -1))

[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]

**round**

round(number[, ndigits]) 함수는 숫자를 입력받아 반올림해 주는 함수이다.

※ [, ndigits]는 ndigits가 있을 수도 있고 없을 수도 있다는 의미이다.

>>> round(4.6)

5

다음과 같이 실수 5.678을 소수점 2자리까지만 반올림하여 표시할 수 있다.

>>> round(5.678, 2)

5.68

round 함수의 두 번째 매개변수는 반올림하여 표시하고 싶은 소수점의 자릿수(ndigits)이다.

**str**

str(object)은 문자열 형태로 객체를 변환하여 돌려주는 함수이다.

>>> str(3)

'3'

>>> str('hi'.upper())

'HI'

**type**

type(object)은 입력값의 자료형이 무엇인지 알려 주는 함수이다.

>>> type("abc")

<class 'str'>

>>> type(open("test", 'w'))

<class '\_io.TextIOWrapper'>

**zip**

zip(\*iterable)은 동일한 개수로 이루어진 자료형을 묶어 주는 역할을 하는 함수이다.

※ 여기서 사용한 \*iterable은 반복 가능(iterable)한 자료형 여러 개를 입력할 수 있다는 의미이다.

잘 이해되지 않는다면 다음 예제를 살펴보자.

>>> list(zip([1, 2, 3], [4, 5, 6]))

[(1, 4), (2, 5), (3, 6)]

>>> list(zip([1, 2, 3], [4, 5, 6], [7, 8, 9]))

[(1, 4, 7), (2, 5, 8), (3, 6, 9)]

>>> list(zip("abc", "def"))

[('a', 'd'), ('b', 'e'), ('c', 'f')]

**외장 함수**

**sys**

sys 모듈은 파이썬 인터프리터가 제공하는 변수와 함수를 직접 제어할 수 있게 해주는 모듈이다.

**pickle**

pickle은 객체의 형태를 그대로 유지하면서 파일에 저장하고 불러올 수 있게 하는 모듈이다. 다음 예는 pickle 모듈의 dump 함수를 사용하여 딕셔너리 객체인 data를 그대로 파일에 저장하는 방법을 보여 준다.

>>> import pickle

>>> f = open("test.txt", 'wb')

>>> data = {1: 'python', 2: 'you need'}

>>> pickle.dump(data, f)

>>> f.close()

다음은 pickle.dump로 저장한 파일을 pickle.load를 사용해서 원래 있던 딕셔너리 객체(data) 상태 그대로 불러오는 예이다.

>>> import pickle

>>> f = open("test.txt", 'rb')

>>> data = pickle.load(f)

>>> print(data)

{2:'you need', 1:'python'}

위 예에서는 딕셔너리 객체를 사용했지만 어떤 자료형이든저장하고 불러올 수 있다.

**os**

OS 모듈은 환경 변수나 디렉터리, 파일 등의 OS 자원을 제어할 수 있게 해주는 모듈이다.

\* 디렉터리 위치 변경하기 - os.chdir

os.chdir를 사용하면 다음과 같이 현재 디렉터리 위치를 변경할 수 있다.

>>> os.chdir("C:\WINDOWS")

\* 디렉터리 위치 돌려받기 - os.getcwd

os.getcwd는 현재 자신의 디렉터리 위치를 돌려준다.

>>> os.getcwd()

'C:\WINDOWS'

\* 시스템 명령어 호출하기 - os.system

시스템 자체의 프로그램이나 기타 명령어를 파이썬에서 호출할 수도 있다. os.system("명령어")처럼 사용한다. 다음은 현재 디렉터리에서 시스템 명령어 dir을 실행하는 예이다.

>>> os.system("dir")

\*실행한 시스템 명령어의 결괏값 돌려받기 - os.popen

os.popen은 시스템 명령어를 실행한 결괏값을 읽기 모드 형태의 파일 객체로 돌려준다.

>>> f = os.popen("dir")

읽어 들인 파일 객체의 내용을 보기 위해서는 다음과 같이 하면 된다.

>>> print(f.read())

\* 기타 유용한 os 관련 함수

함수 설명

os.mkdir(디렉터리) 디렉터리를 생성한다.

os.rmdir(디렉터리) 디렉터리를 삭제한다.단, 디렉터리가 비어있어야 삭제가 가능하다.

os.unlink(파일) 파일을 지운다.

os.rename(src, dst) src라는 이름의 파일을 dst라는 이름으로 바꾼다.

**shutil**

shutil은 파일을 복사해 주는 파이썬 모듈이다.

다음 예시는 src라는 이름의 파일을 dst로 복사한다. 만약 dst가 디렉터리 이름이라면 src라는 파일 이름으로 dst 디렉터리에 복사하고 동일한 파일 이름이 있을 경우에는 덮어쓴다.

>>> import shutil

>>> shutil.copy("src.txt", "dst.txt")

위 예를 실행해 보면 src.txt 파일과 동일한 내용의 파일이 dst.txt로 복사되는 것을 확인할 수 있다.

**glob**

가끔 파일을 읽고 쓰는 기능이 있는 프로그램을 만들다 보면 특정 디렉터리에 있는 파일 이름 모두를 알아야 할 때가 있다. 이럴 때 사용하는 모듈이 바로 glob이다.

\* 디렉터리에 있는 파일들을 리스트로 만들기 - glob(pathname)

glob 모듈은 디렉터리 안의 파일들을 읽어서 돌려준다. \*, ? 등 메타 문자를 써서 원하는 파일만 읽어 들일 수도 있다.

다음은 C:/doit 디렉터리에 있는 파일 중 이름이 mark로 시작하는 파일을 모두 찾아서 읽어들이는 예이다.

>>> import glob

>>> glob.glob("c:/doit/mark\*")

['c:/doit\\marks1.py', 'c:/doit\\marks2.py', 'c:/doit\\marks3.py']

>>>

**tempfile**

파일을 임시로 만들어서 사용할 때 유용한 모듈이 바로 tempfile이다. tempfile.mkstemp()는 중복되지 않는 임시 파일의 이름을 무작위로 만들어서 돌려준다.

>>> import tempfile

>>> filename = tempfile.mkstemp()

>>> filename

'C:\WINDOWS\TEMP\~-275151-0'

tempfile.TemporaryFile()은 임시 저장 공간으로 사용할 파일 객체를 돌려준다. 이 파일은 기본적으로 바이너리 쓰기 모드(wb)를 갖는다. f.close()가 호출되면 이 파일 객체는 자동으로 사라진다.

>>> import tempfile

>>> f = tempfile.TemporaryFile()

>>> f.close()

**time**

\* time.time

time.time()은 UTC(Universal Time Coordinated 협정 세계 표준시)를 사용하여 현재 시간을 실수 형태로 돌려주는 함수이다. 1970년 1월 1일 0시 0분 0초를 기준으로 지난 시간을 초 단위로 돌려준다.

>>> import time

>>> time.time()

988458015.73417199

\* time.localtime

time.localtime은 time.time()이 돌려준 실수 값을 사용해서 연도, 월, 일, 시, 분, 초, ... 의 형태로 바꾸어 주는 함수이다.

>>> time.localtime(time.time())

time.struct\_time(tm\_year=2013, tm\_mon=5, tm\_mday=21, tm\_hour=16,

tm\_min=48, tm\_sec=42, tm\_wday=1, tm\_yday=141, tm\_isdst=0)

time.asctime

\* time.localtime에 의해서 반환된 튜플 형태의 값을 인수로 받아서 날짜와 시간을 알아보기 쉬운 형태로 돌려주는 함수이다.

>>> time.asctime(time.localtime(time.time()))

'Sat Apr 28 20:50:20 2001'

\* time.ctime

time.asctime(time.localtime(time.time()))은 time.ctime()을 사용해 간편하게 표시할 수 있다. asctime과 다른 점은 ctime은 항상 현재 시간만을 돌려준다는 점이다.

>>> time.ctime()

'Sat Apr 28 20:56:31 2001'

\*time.strftime

time.strftime('출력할 형식 포맷 코드', time.localtime(time.time()))

strftime 함수는 시간에 관계된 것을 세밀하게 표현하는 여러 가지 포맷 코드를 제공한다.

\* 시간에 관계된 것을 표현하는 포맷 코드

포맷코드 설명 예

%a 요일 줄임말 Mon

%A 요일 Monday

%b 달 줄임말 Jan

%B 달 January

%c 날짜와 시간을 출력함 06/01/01 17:22:21

%d 날(day) [01,31]

%H 시간(hour)-24시간 출력 형태 [00,23]

%I 시간(hour)-12시간 출력 형태 [01,12]

%j 1년 중 누적 날짜 [001,366]

%m 달 [01,12]

%M 분 [01,59]

%p AM or PM AM

%S 초 [00,59]

%U 1년 중 누적 주-일요일을 시작으로 [00,53]

%w 숫자로 된 요일 [0(일요일),6]

%W 1년 중 누적 주-월요일을 시작으로 [00,53]

%x 현재 설정된 로케일에 기반한 날짜 출력 06/01/01

%X 현재 설정된 로케일에 기반한 시간 출력 17:22:21

%Y 년도 출력 2001

%Z 시간대 출력 대한민국 표준시

%% 문자 %

%y 세기부분을 제외한 년도 출력 01

다음은 time.strftime을 사용하는 예이다.

>>> import time

>>> time.strftime('%x', time.localtime(time.time()))

'05/01/01'

>>> time.strftime('%c', time.localtime(time.time()))

'05/01/01 17:22:21'

\* time.sleep

#sleep1.py

import time

for i in range(10):

print(i)

time.sleep(1)

위 예는 1초 간격으로 0부터 9까지의 숫자를 출력한다. 위 예에서 볼 수 있듯이 time.sleep 함수의 인수는 실수 형태를 쓸 수 있다. 즉 1이면 1초, 0.5면 0.5초가 되는 것이다.

**calendar**

calendar는 파이썬에서 달력을 볼 수 있게 해주는 모듈이다.

calendar.calendar(연도)로 사용하면 그해의 전체 달력을 볼 수 있다. 결괏값은 달력이 너무 길어 생략하겠다.

>>> import calendar

>>> print(calendar.calendar(2015)) 혹은 calendar.prcal(2015)

\* 다음 예는 2015년 12월의 달력만 보여 준다.

>>> calendar.prmonth(2015, 12)

\* calendar.weekday

calendar 모듈의 또 다른 유용한 함수를 보자. weekday(연도, 월, 일) 함수는 그 날짜에 해당하는 요일 정보를 돌려준다. 월요일은 0, 화요일은 1, 수요일은 2, 목요일은 3, 금요일은 4, 토요일은 5, 일요일은 6이라는 값을 돌려준다.

>>> calendar.weekday(2015, 12, 31)

3

\* calendar.monthrange

monthrange(연도, 월) 함수는 입력받은 달의 1일이 무슨 요일인지와 그 달이 며칠까지 있는지를 튜플 형태로 돌려준다.

>>> calendar.monthrange(2015,12)

(1, 31)

위 예는 2015년 12월 1일은 화요일이고, 이 달은 31일까지 있다는 것을 보여 준다.

날짜와 관련된 프로그래밍을 할 때 위 2가지 함수는 매우 유용하게 사용된다.

**random**

\* 다음은 0.0에서 1.0 사이의 실수 중에서 난수 값을 돌려주는 예를 보여 준다.

>>> import random

>>> random.random()

0.53840103305098674

\* 다음 예는 1에서 10 사이의 정수 중에서 난수 값을 돌려준다.

>>> random.randint(1, 10)

6

\* random.choice 함수는 입력으로 받은 리스트에서 무작위로 하나를 선택하여 돌려준다.

\* 리스트의 항목을 무작위로 섞고 싶을 때는 random.shuffle 함수를 사용하면 된다.

**webbrowser**

webbrowser는 자신의 시스템에서 사용하는 기본 웹 브라우저를 자동으로 실행하는 모듈이다. 다음 예제는 웹 브라우저를 자동으로 실행하고 해당 URL인 google.com으로 가게 해 준다.

>>> import webbrowser

>>> webbrowser.open("http://google.com")

webbrowser의 open 함수는 웹 브라우저가 이미 실행된 상태라면 입력 주소로 이동한다. 만약 웹 브라우저가 실행되지 않은 상태라면 새로 웹 브라우저를 실행한 후 해당 주소로 이동한다.

\* open\_new 함수는 이미 웹 브라우저가 실행된 상태이더라도 새로운 창으로 해당 주소가 열리게 한다.

>>> webbrowser.open\_new("http://google.com")

**[스레드를 다루는 threading 모듈]**

# thread\_test.py

import time

def long\_task(): # 5초의 시간이 걸리는 함수

for i in range(5):

time.sleep(1) # 1초간 대기한다.

print("working:%s\n" % i)

print("Start")

for i in range(5): # long\_task를 5회 수행한다.

long\_task()

print("End")

long\_task 함수는 수행하는 데 5초의 시간이 걸리는 함수이다. 위 프로그램은 이 함수를 총 5번 반복해서 수행하는 프로그램이다. 이 프로그램은 5초가 5번 반복되니 총 25초의 시간이 걸린다.

하지만 앞에서 설명했듯이 스레드를 사용하면 5초의 시간이 걸리는 long\_task 함수를 동시에 실행할 수 있으니 시간을 줄일 수 있다.

다음과 같이 프로그램을 수정해 보자.

# thread\_test.py

import time

import threading # 스레드를 생성하기 위해서는 threading 모듈이 필요하다.

def long\_task():

for i in range(5):

time.sleep(1)

print("working:%s\n" % i)

print("Start")

threads = []

for i in range(5):

t = threading.Thread(target=long\_task) # 스레드를 생성한다.

threads.append(t)

for t in threads:

t.start() # 스레드를 실행한다.

print("End")

이와 같이 프로그램을 수정하고 실행해 보면 25초 걸리던 작업이 5초 정도에 수행되는 것을 확인할 수 있다. threading.Thread를 사용하여 만든 스레드 객체가 동시 작업을 가능하게 해 주기 때문이다.

하지만 위 프로그램을 실행해 보면 "Start"와 "End"가 먼저 출력되고 그 이후에 스레드의 결과가 출력되는 것을 확인할 수 있다. 그리고 프로그램이 정상 종료되지 않는다. 우리가 기대하는 것은 "Start"가 출력되고 그다음에 스레드의 결과가 출력된 후 마지막으로 "End"가 출력되는 것이다.

이 문제를 해결하기 위해서는 다음과 같이 프로그램을 수정해야 한다.

# thread\_test.py

import time

import threading

def long\_task():

for i in range(5):

time.sleep(1)

print("working:%s\n" % i)

print("Start")

threads = []

for i in range(5):

t = threading.Thread(target=long\_task)

threads.append(t)

for t in threads:

t.start()

for t in threads:

t.join() # join으로 스레드가 종료될때까지 기다린다.

print("End")

스레드의 join 함수는 해당 스레드가 종료될 때까지 기다리게 한다. 따라서 위와 같이 수정하면 우리가 원하던 출력을 보게 된다.