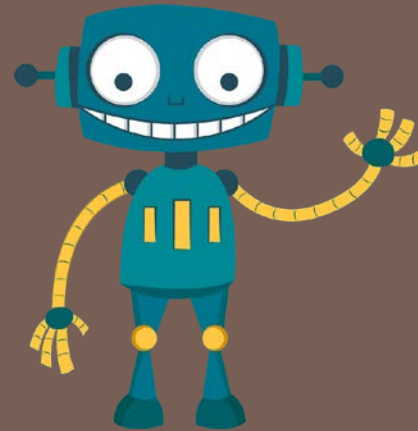
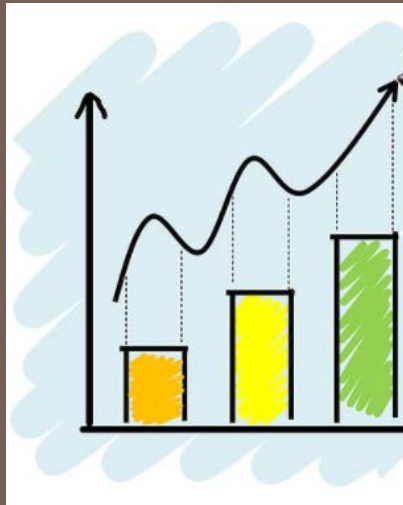


# 파이썬 익스프레스



11 장 내장함수, 람다식, 제너레이터, 모듈

# 학습 목표

- 파이썬에 내장된 내장 함수들을 살펴본다.
- 람다식을 살펴본다.
- 제너레이터를 사용하여 반복 가능한 객체를 작성할 수 있다.
- 모듈의 개념을 살펴본다.
- 유용한 모듈을 사용할 수 있다.



# 내장 함수

- 파이썬 인터프리터에는 항상 사용할 수 있는 많은 함수가 준비되어 있다. 이러한 함수를 **내장 함수**라고 한다.

내장 함수				
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()



# abs() 함수

- **abs()** 함수는 숫자의 절대값을 반환하는 데 사용된다.

```
>>> i = -20
```

```
>>> abs(i)
```

```
20
```

```
>>> f = -30.92
```

```
>>> abs(f)
```

```
30.92
```

# all() 함수

- **all() 함수**는 시퀀스(리스트나 딕셔너리 등)를 받아서, 시퀀스의 모든 항목이 참이면 **True**를 반환한다.

```
>>> mylist = [1, 3, 4, 6]           # 모든 값이 참이다.  
>>> all(mylist)  
True
```

```
>>> mylist = [1, 3, 4, 0]          # 하나의 값이 거짓이다.  
>>> all(mylist)  
False
```

```
>>> mylist = [True, 0, False, 0]  # 하나의 값만 참이다.  
>>> all(mylist)  
False
```

# any() 함수

- **any() 함수**는 시퀀스 객체에 있는 한 개의 항목이라도 참인 경우 참을 반환한다.

```
>>> mylist = [0, 1, 2, 3]
```

```
>>> any(mylist)
```

```
True
```

```
>>> mylist = [0, False]
```

```
>>> any(mylist)
```

```
False
```

# eval() 함수

- **eval() 함수**는 전달된 수식을 구문 분석하고 프로그램 내에서 수식을 실행한다.

```
>>> x = 10
```

```
>>> eval('x + 1')
```

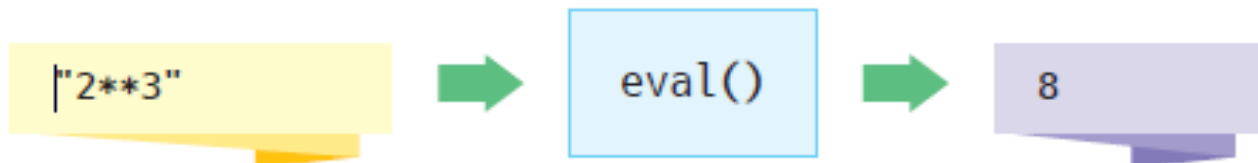
```
11
```

```
>>> exp = input("파이썬의 수식을 입력하시오: ")
```

```
파이썬의 수식을 입력하시오: 2**10
```

```
>>> eval(exp)
```

```
1024
```



# sum() 함수

- **sum()** 함수는 리스트에 존재하는 항목들을 전부 더하여 합계를 반환한다.

```
>>> sum([1, 2, 3])  
6
```

```
>>> sum([1, 2, 3], 20)  
26
```



# len() 함수

- **len()** 함수는 객체의 길이를 계산하여 반환하는 함수이다

```
>>> len("All's well that ends well. ")  
27
```

```
>>> len([1, 2, 3, 4, 5])  
5
```

# list() 함수

- **list()**는 리스트를 생성하는 함수이다.

```
>>> s = 'abcdefg'
```

```
>>> list(s)
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

```
>>> t = (1, 2, 3, 4, 5, 6)
```

```
>>> list(t)
```

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

# map() 함수

- **map()** 함수는 반복 가능한 객체(리스트, 튜플 등)의 각 항목에 주어진 함수를 적용한 후, 결과를 반환한다

```
def square(n):
```

```
    return n*n
```

```
mylist = [1, 2, 3, 4, 5]
```

```
result = list( map(square, mylist) )
```

```
print(result)
```

```
[1, 4, 9, 16, 25]
```

# dir() 함수

- **dir()**은 객체가 가지고 있는 변수나 함수를 보여 준다.

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

```
['__add__',
```

```
'__class__',
```

```
'__contains__',
```

```
'__delattr__',
```

```
...
```

# max(), min() 함수

- **max()** 함수는 리스트나 튜플, 문자열에서 가장 큰 항목을 반환한다.
- **min()** 함수는 리스트나 튜플, 문자열에서 가장 작은 항목을 반환한다.

```
>>> values = [ 1, 2, 3, 4, 5]
```

```
>>> max(values)
```

```
5
```

```
>>> min(values)
```

```
1
```

# enumerate() 함수

- **enumerate()**는 시퀀스 객체를 입력 받아, 열거형 (**enumerate**) 객체를 반환한다.

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
```

```
>>> list( enumerate(seasons) )
```

```
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
```

```
>>> list( enumerate(seasons, start=1) )
```

```
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

# filter() 함수

- **filter()** 함수는 특정 조건을 만족하는 요소만을 뽑는다.

```
def myfilter(x):
```

```
    return x > 3
```

```
result = filter( myfilter, (1, 2, 3, 4, 5, 6))
```

```
print( list(result) )
```

```
[4, 5, 6]
```

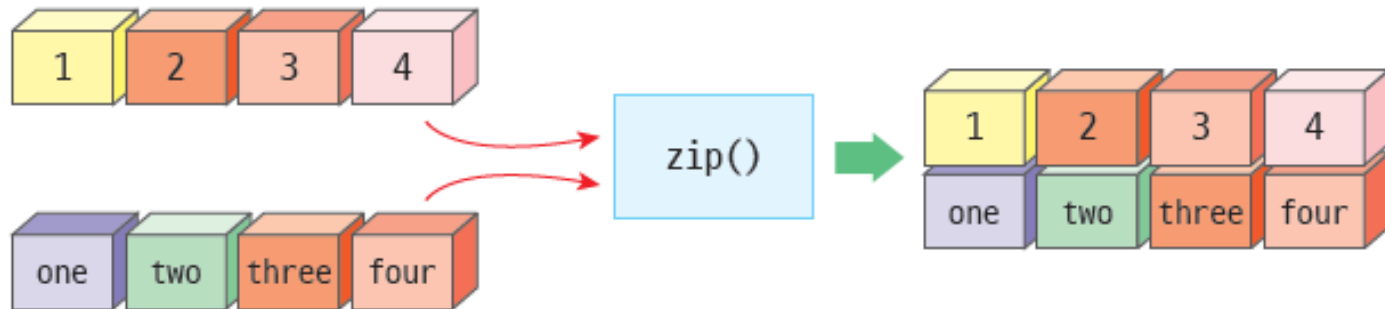
# zip() 함수

- **zip() 함수**는 2개의 자료형을 하나로 묶어주는 함수이다.

```
>>> numbers = [1, 2, 3, 4]
```

```
>>> slist = ['one', 'two', 'three', 'four']
```

```
>>> list( zip(numbers, slist) )  
[(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
```





# Lab: 내장 함수 예제

- 우리는 파티에 적합한 양의 음식을 구입하기 위해서는 총 인원이 몇 명인지 알아야 한다.

```
>>> invitations = ["Kim", "Lee", "Park", "Choi"] # 참석자 이름
>>> persons = [1, 3, 0, 6] # 동반자 수
>>> sum(persons)
10
```

파티에 한 사람이라도 오는 지를 확인해보자. 이것은 **any()** 함수로 가능하다.

```
>>> any(persons)
True
```

이번에는 모든 초대받은 그룹이 전부 오는 지는 확인해보자. 이것은 **all()** 함수로 가능하다.

```
>>> all(persons)
False
```

가장 많이 오는 그룹에는 몇 사람이나 있는지를 알아보자. **max()**로 가능하다.

```
>>> max(persons)
6
```

# 정렬과 탐색

- 파이썬 리스트는 **sort()**라는 메소드를 가지고 이 메소드는 리스트를 정렬된 상태로 변경한다.
- **sorted()**라는 내장 함수는 반복 가능한 객체로부터 정렬된 리스트를 생성한다.

```
>>> sorted([4, 2, 3, 5, 1])
```

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

```
>>> myList = [4, 2, 3, 5, 1]
```

```
>>> myList.sort()          # myList가 바뀜
```

```
>>> myList
```

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



# key 매개변수

- 정렬을 하다 보면 정렬에 사용되는 키를 개발자가 변경해주어야 하는 경우가 종종 있다.
- 아래 예는 단어 비교 전에 `lower()` 실행하여 단어를 소문자로 변경함

```
>>> sorted("The health know not of their health, but only the sick".split(),  
key=str.lower)  
['but', 'health', 'health,', 'know', 'not', 'of', 'only', 'sick', 'The', 'the', 'their']
```

➤ key 매개변수를 빼고 실행하면?

# key 매개변수

```
students = [  
    ('홍길동', 3.9, 20210303),  
    ('김철수', 3.0, 20210302),  
    ('최자영', 4.3, 20210301),  
]  
  
# 학번을 기준으로 정렬  
print( sorted( students, key=lambda student: student[2]) )
```

```
[('최자영', 4.3, 20210301), ('김철수', 3.0, 20210302), ('홍길동', 3.9, 20210303)]
```

# key를 이용한 정렬 예제

```
class Student:
    def __init__(self, name, grade, number):
        self.name = name
        self.grade = grade
        self.number = number
    def __repr__(self):
        return repr((self.name, self.grade, self.number))

students = [
    Student('홍길동', 3.9, 20210303),
    Student('김철수', 3.0, 20210302),
    Student('최자영', 4.3, 20210301),
]
print( sorted(students, key=lambda student: student.number) )
```

```
[('최자영', 4.3, 20210301), ('김철수', 3.0, 20210302), ('홍길동', 3.9, 20210303)]
```

# 오름차순 정렬과 내림차순 정렬

- 기본적으로 오름차순 정렬 실행
- 내림차순 정렬
  - ▶ **reverse** 매개변수를 **True**로 설정

```
>>> sorted(students, key=lambda student: student.number, reverse=True)
[('홍길동', 3.9, 20210303), ('김철수', 3.0, 20210302), ('최자영', 4.3, 20210301)]
```

# Lab: 키를 이용한 정렬 예제

- 간단하게 사람들의 이름과 나이만 저장하는 주소록을 작성하자.
- 사람을 **Person** 이라는 클래스로 나타낸다.
- **Person 클래스**는 다음과 같은 인스턴스 변수를 가진다.
  - **name** – 이름 (문자열)
  - **age** – 나이 (정수형)
- 나이 순으로 정렬하여 보여주는 프로그램을 작성하자.
  - 정렬의 기준은 사람의 나이이다.

[<이름: 홍길동, 나이: 20>, <이름: 김철수, 나이: 35>, <이름: 최자영, 나이: 38>]

# Lab: 키를 이용한 정렬 예제

```
class Person(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __repr__(self):
        return "<이름: %s, 나이: %s>" % (self.name, self.age)

def keyAge(person):
    return person.age

people = [Person("홍길동", 20), Person("김철수", 35), Person("최자영", 38)]
print( sorted(people, key = keyAge) )
```

```
[<이름: 홍길동, 나이: 20>, <이름: 김철수, 나이: 35>, <이름: 최자영, 나이: 38>]
```



# 람다식

- 람다식은 이름은 없고 몸체만 있는 함수이다.
- 람다식은 **lambda** 키워드로 만들어진다.
- 람다식은 딱 한 번 사용하는 함수를 만드는데 사용된다.

## Syntax: 람다식 정의

**형식**    lambda 매개 변수들: 수식

**예**    lambda x, y: x+y;

매개 변수

함수의 몸체

# 예제

```
f = lambda x, y: x+y
```

```
print( "정수의 합 :", f( 10, 20 ))
```

```
print( "정수의 합 :", f( 20, 20 ))
```



```
정수의 합 : 30  
정수의 합 : 40
```

```
def get_sum(x, y):  
    return x+y
```

```
print( "정수의 합 :", get_sum( 10, 20 ))
```

```
print( "정수의 합 :", get_sum( 20, 20 ))
```

# 콜백 함수

```
from tkinter import *

window = Tk()

btn1 = Button(window, text="1 출력", command=lambda: print(1, "버튼이 클릭"))
btn1.pack(side=LEFT)
btn2 = Button(window, text="2 출력", command=lambda: print(2, "버튼이 클릭"))
btn2.pack(side=LEFT)

quitBtn = Button(window, text="QUIT", fg="red", command=quit)
quitBtn.pack(side=LEFT)

mainloop()
```

2 버튼이 클릭  
1 버튼이 클릭  
2 버튼이 클릭



# map() 함수와 람다식

```
list_a = [ 1, 2, 3, 4, 5 ]
```

```
f = lambda x : 2*x
```

```
result = map(f, list_a)
```

```
print( list(result) )
```

```
[2, 4, 6, 8, 10]
```

# filter() / sorted() 함수와 람다식

```
list_a = [1, 2, 3, 4, 5, 6]  
result = filter( lambda x : x % 2 == 0, list_a )  
print( list(result) )
```

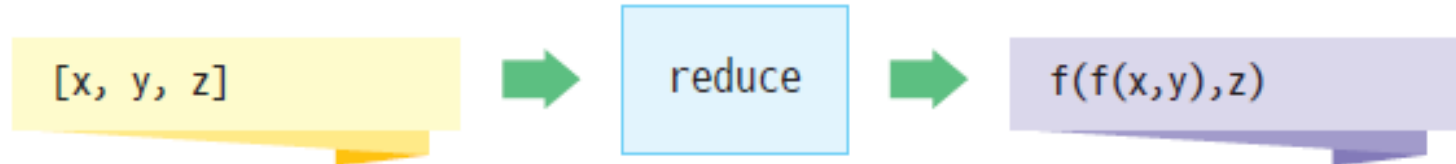
```
[2, 4, 6]
```

```
data = [(3, 100), (1, 500), (7, 300), (6, 400)]  
print( sorted(data, key=lambda item: item[0]) )  
# print(data)    ??
```

```
[(1, 500), (3, 100), (6, 400), (7, 300)]
```

# reduce() 함수와 람다식

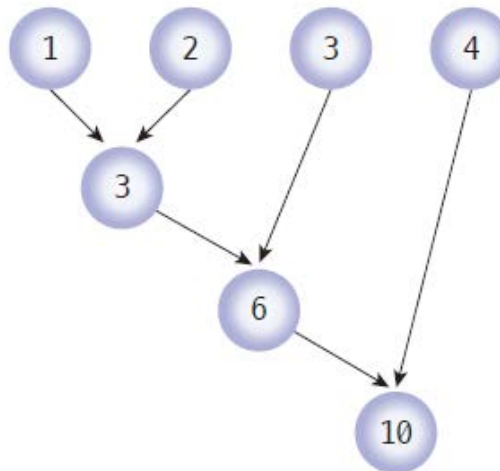
- `reduce(func, seq)` 함수는 `func()` 함수를 시퀀스 `seq`에 연속적으로 적용하여 단일 값을 반환한다.



# 예제

```
import functools  
result = functools.reduce(lambda x,y: x+y, [1, 2, 3, 4])  
print(result)
```

10



# Lab: 람다식으로 온도 변환하기

- 람다식과 `map()` 함수를 사용하여 여러 개의 화씨 온도를 섭씨 온도로 변환하는 명령문을 작성해보자.
- 람다식을 사용하지 않는 버전은 다음과 같다.

```
def celsius(T):  
    return (5.0/9.0)*(T-32.0)  
  
f_temp = [0, 10, 20, 30, 40, 50]  
  
c_temp = map(celsius, f_temp)  
print( list(c_temp) )
```

```
[-17.777777777777778, -12.222222222222223, -6.666666666666667, -  
1.1111111111111112, 4.444444444444445, 10.0]
```



# Sol: 람다식으로 온도 변환하기

```
##  
#       이 프로그램은 람다식을 이용하여 온도를 변환한다.  
#  
  
f_temp = [0, 10, 20, 30, 40, 50]  
c_temp = map(lambda x: (5.0/9.0)*(x-32.0), f_temp)  
print( list(c_temp) )
```

```
[-17.777777777777778, -12.222222222222223, -6.666666666666667, -  
1.1111111111111112, 4.444444444444445, 10.0]
```

# Lab: 람다식으로 데이터 처리하기

- 아래의 데이터를 처리하여서 (주문번호, 총주문금액)이 저장된 튜플의 리스트를 반환하는 프로그램을 작성하라

주문 번호	물품	수량	품목당 가격
1	"재킷"	5	120000
2	"셔츠"	6	24000
3	"바지"	3	50000
4	"코트"	6	300000

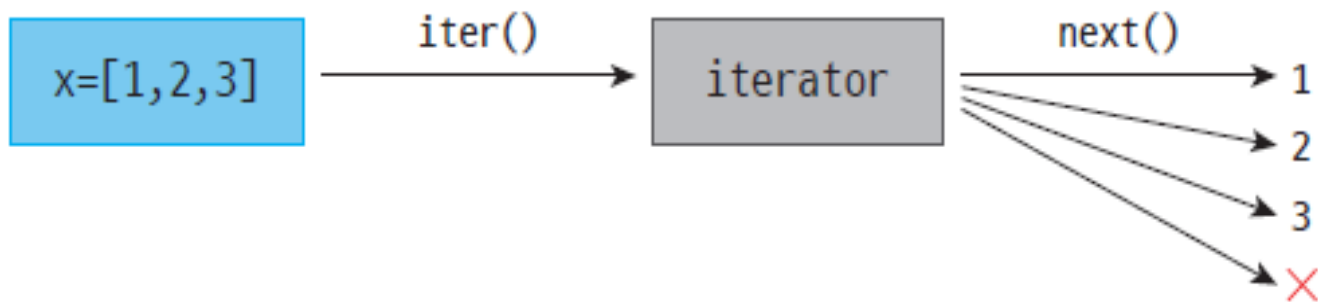
```
[('1', 600000), ('2', 144000), ('3', 150000), ('4', 1800000)]
```

# Sol:

```
##  
#       이 프로그램은 람다식을 이용하여 주문을 처리한다.  
#  
orders = [ ["1", "재킷", 5, 120000],  
            ["2", "셔츠", 6, 24000],  
            ["3", "바지", 3, 50000],  
            ["4", "코트", 6, 300000] ]  
  
result = list( map( lambda x: (x[0], x[2] * x[3]), orders ) )  
print(result)
```

# 이터레이터

- 파이썬에서는 **for** 루프와 함께 사용할 수 있는 여러 종류의 객체가 있으며 이들 객체는 **이터러블 객체 (iterable object)**이라고 불린다.



# 객체가 이터러블 객체가 되려면

- `__iter__()`은 이터러블 객체 자신을 반환한다.
- `__next__()`은 다음 반복을 위한 값을 반환한다.
  - 만약 더 이상의 값이 없으면 **StopIteration** 예외를 발생하면 된다.

# 예제

```
class MyCounter(object):
    # 생성자 메소드를 정의한다.
    def __init__(self, low, high):
        self.current = low
        self.high = high

    # 이터레이터 객체로서 자신을 반환한다.
    def __iter__(self):
        return self

    def __next__(self):
        # current가 high 보다 크면 StopIteration 예외를 발생한다.
        # current가 high 보다 작으면 다음 값을 반환한다.
        if self.current > self.high:
            raise StopIteration           # 예외를 발생시킴
        else:
            self.current += 1
            return self.current - 1
```

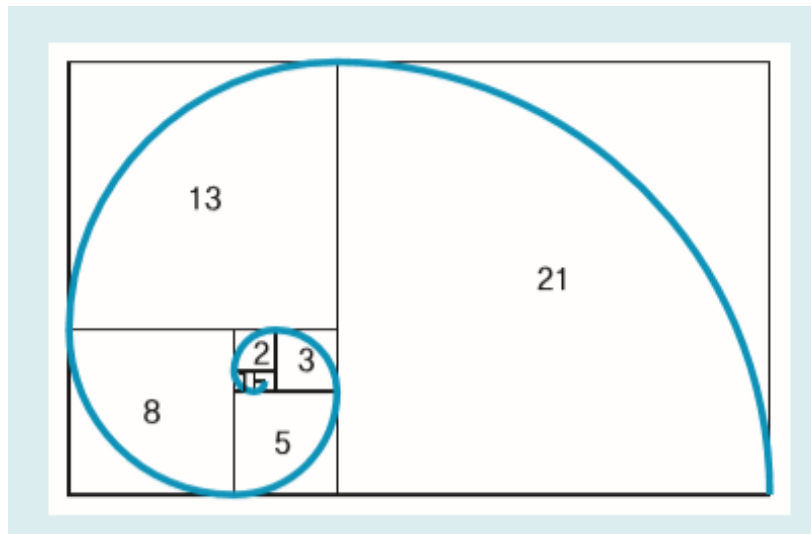
# 예제

```
c = MyCounter(1, 10)  
for i in c:  
    print(i, end=' ')
```

```
1 2 3 4 5 6 7 8 9 10
```

# Lab: 피보나치 이터레이터

- 피보나치 수열이란 앞의 두 수의 합이 바로 뒤의 수가 되는 수열을 의미한다.
- 피보나치 수열의 수들을 생성하는 이터레이터 클래스를 정의해보자.



1 1 2 3 5 8 13 21 34



# Solution

```
class Fiblterator:
```

```
    def __init__(self, a=1, b=0, maxValue=50):
```

```
        self.a = a
```

```
        self.b = b
```

```
        self.maxValue = maxValue
```

```
    def __iter__(self):
```

```
        return self
```

```
    def __next__(self):
```

```
        n = self.a + self.b
```

```
        if n > self.maxValue:
```

```
            raise StopIteration
```

```
        self.a = self.b
```

```
        self.b = n
```

```
        return n
```

```
for i in Fiblterator(): # Fiblterator 객체로부터 이터레이터 객체 받음
    print(i, end=" ")
```

# 제너레이터

- 제너레이터(**generators**)는 키워드 **yield**를 사용하여 **함수로부터** 이터레이터를 생성하는 하나의 방법이다.

```
def myGenerator():
```

```
    yield 'first'
```

```
    yield 'second'
```

```
    yield 'third'
```

`__next()__`

```
def myGenerator():
```

```
    yield 'first'
```

```
    yield 'second'
```

```
    yield 'third'
```

'first'

'second'

'third'

# 예제

```
def myGenerator():
```

```
    yield 'first'
```

```
    yield 'second'
```

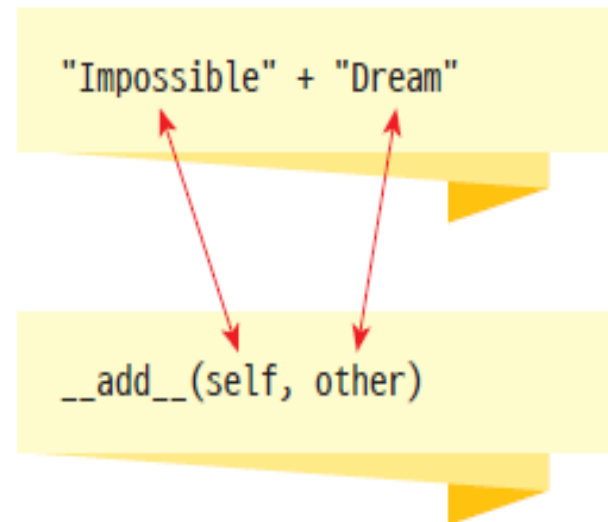
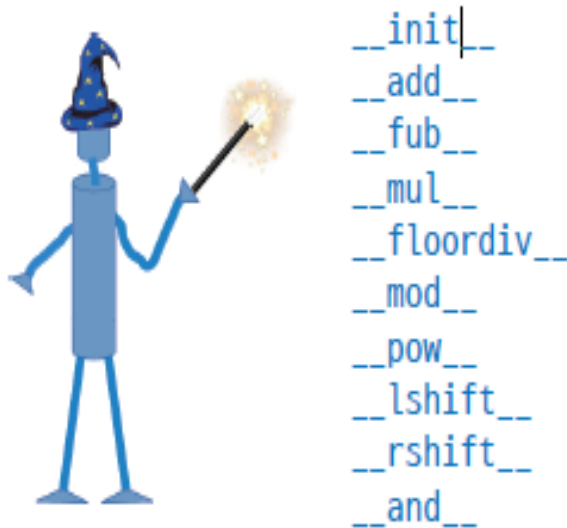
```
    yield 'third'
```

```
for word in myGenerator():    # myGenerator()함수로부터 이터레이터 생성
    print(word)
```

```
first
second
third
```

# 연산자 오버로딩

- 연산자를 메소드로 정의하는 것을 **연산자 오버로딩 (operator overloading)**이라고 한다.



# 오버로딩할 수 있는 연산자

연산자	수식예	내부적인 함수 호출
덧셈	$x + y$	<code>x.__add__(y)</code>
뺄셈	$x - y$	<code>x.__sub__(y)</code>
곱셈	$x * y$	<code>x.__mul__(y)</code>
지수	$x ** y$	<code>x.__pow__(y)</code>
나눗셈(실수)	$x / y$	<code>x.__truediv__(y)</code>
나눗셈(정수)	$x // y$	<code>x.__floordiv__(y)</code>
나머지	$x \% y$	<code>x.__mod__(y)</code>
작음	$x < y$	<code>x.__lt__(y)</code>
작거나 같음	$x \leq y$	<code>x.__le__(y)</code>
같음	$x == y$	<code>x.__eq__(y)</code>
같지 않음	$x != y$	<code>x.__ne__(y)</code>
큼	$x > y$	<code>x.__gt__(y)</code>
크거나 같음	$x \geq y$	<code>x.__ge__(y)</code>

# 예제

```
>>> s1="Impossible "  
>>> s2="Dream"  
>>> s3 = s1.__add__(s2)  
>>> s3  
'Impossible Dream'  
>>>
```

```
>>> class Point:  
    def __init__(self, x = 0, y = 0):  
        self.x = x  
        self.y = y  
  
>>> p1 = Point(1, 2)  
>>> p2 = Point(3, 4)  
>>> p1 + p2  
...  
TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

# 예제

```
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)

    def __str__(self):
        return 'Point(' + str(self.x) + ', ' + str(self.y) + ')'

p1 = Point(1, 2)
p2 = Point(3, 4)
print(p1 + p2)
```

# Lab: Book 클래스

- 책을 나타내는 **Book** 클래스를 작성한다.
- **Book** 클래스 내부에 **\_\_gt\_\_()** 함수를 정의하여서 **Book** 클래스의 객체가 크지를 비교(>) 할 수 있게 한다.

```
book1 = Book('Magic of Python', 600)
book2 = Book('Master of Python', 700)
print(book1 > book2)
```

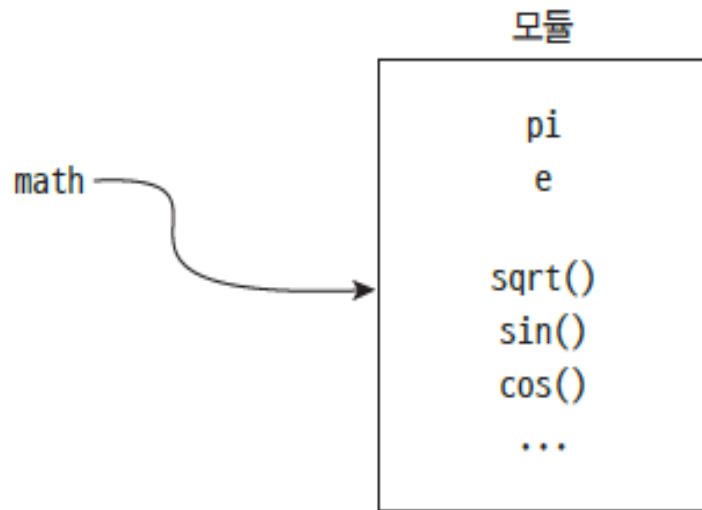


# Solution

```
##  
# 이 프로그램은 연산자 중복을 사용하여 Book 객체의 비교 연산 > 를 구현  
#  
class Book:  
    title = "  
    pages = 0  
  
    def __init__(self, title="", pages=0):  
        self.title = title  
        self.pages = pages  
  
    def __str__(self):  
        return self.title  
  
    def __gt__(self, other):    # Book 객체의 > 연산 구현  
        return self.pages > other.pages
```

# 모듈 (Module)

- 파이썬에서 **모듈(module)**이란 함수나 변수 또는 클래스들을 모아 놓은 파일이다.

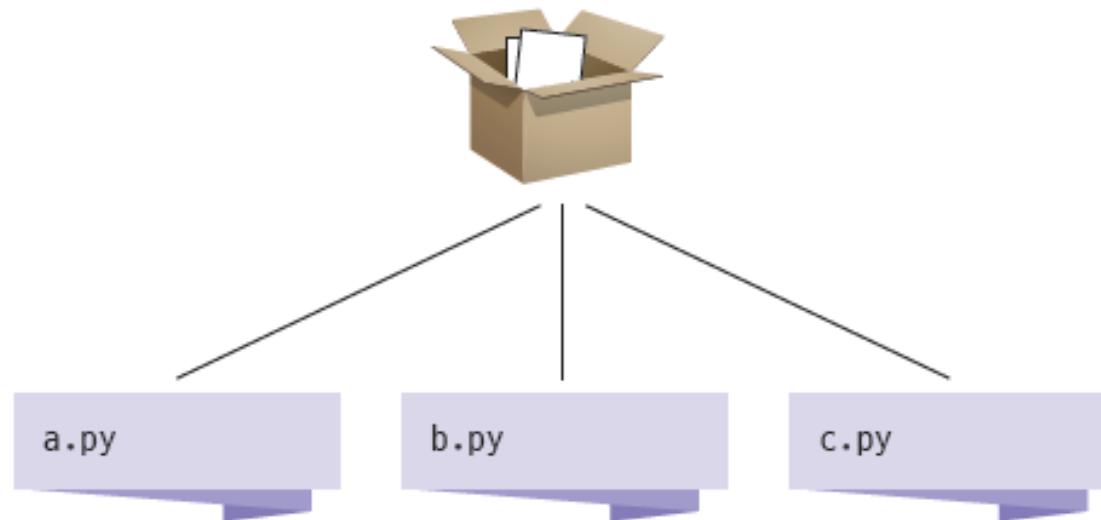


모듈은 파이썬의 문장들이  
저장된 파일입니다.



# 모듈

- 파이썬 프로그램이 길어지면, 유지 보수를 쉽게 하기 위해 여러 개의 **파일로 분할** 할 수 있다.
- 파일로 저장해 놓으면 한번 작성한 함수를 복사하지 않고, 여러 프로그램에서 재사용할 수 있다.



# 모듈 작성하기

## fibonacci.py

# 피보나치 수열 모듈

```
def fib(n):          # 피보나치 수열 출력
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()
```

```
def fib2(n):         # 피보나치 수열을 리스트로 반환
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

# 모듈 사용하기

```
>>> import fibo
```

```
>>> fibo.fib(1000)
```

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

```
>>> fibo.fib2(100)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

# from 모듈 import 함수

- 만약 `fibonacci.fib()`와 같이 함수를 사용할 때마다 모듈의 이름을 적어주는 것이 귀찮다면 다음과 같이 “**from 모듈 import 함수**” 문장을 사용하여도 된다.

```
>>> from fibo import fib
```

```
>>> fib(1000)
```

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

```
>>> from fibo import *
```

```
>>> fib(500)
```

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

# 모듈의 별칭

```
import mymodule as lib
```

```
...
```

```
lib.func(100)
```

# 모듈 실행하기

```
C:> python fibonacci.py <arguments>
```

```
...  
if __name__ == "__main__":    # 모듈을 명령창에서 직접 실행할 수 있게  
    import sys  
    fib( int(sys.argv[1]) )
```

```
C:> python fibonacci.py 50  
1 1 2 3 5 8 13 21 34
```



# 모듈 탐색 경로

- 1) 입력 스크립트가 있는 디렉토리(파일이 지정되지 않으면 현재 디렉토리)
- 2) PYTHONPATH 환경 변수
- 3) 설치에 의존하는 디폴트값

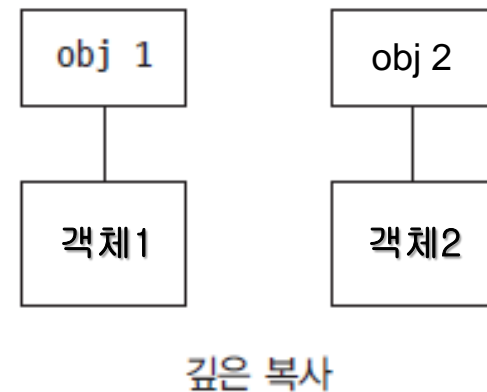
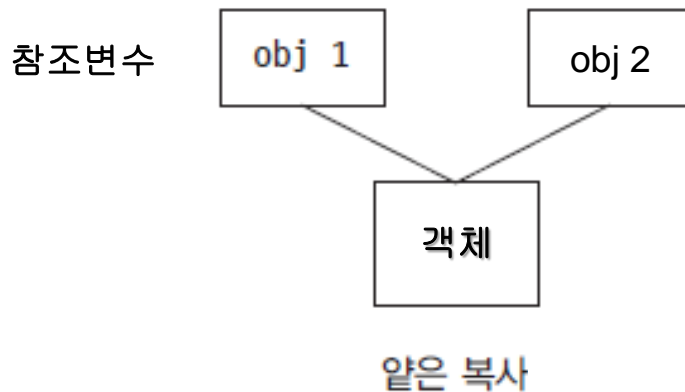
# 유용한 모듈

- 프로그래밍에서 중요한 하나의 원칙은 이전에 개발된 코드를 적극적으로 **재활용**하자는 것



# copy 모듈

- 얽은 복사(shallow copy)
  - 객체의 참조값만 복사되고 객체 자체는 복사되지 않는다.
- 깊은 복사(deep copy)
  - 객체까지 복사된다.

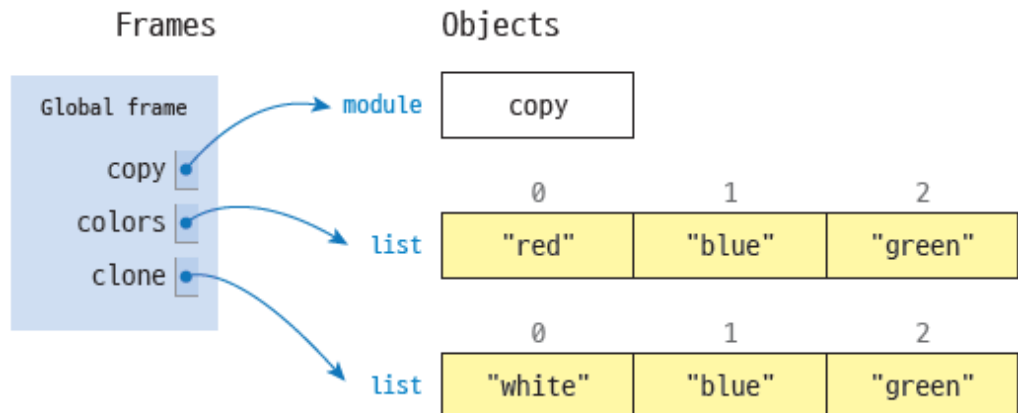


# copy 모듈

```
import copy
colors = ["red", "blue", "green"]
clone = copy.deepcopy(colors)
```

```
clone[0] = "white"
print(colors)
print(clone)
```

```
['red', 'blue', 'green']
['white', 'blue', 'green']
```



# random 모듈

```
>>> import random
>>> print(random.randint(1, 6))
6
>>> print(random.randint(1, 6))
3
```

```
>>> import random
>>> print(random.random()*100)      # 0 ~1 사이의 난수 선택
81.1618515880431
```

```
>>> myList = [ "red", "green", "blue" ]
>>> random.choice(myList)
'blue'
```

# random 모듈

```
>>> myList = [ [x] for x in range(10) ]
>>> random.shuffle(myList)      # 리스트 항목을 랜덤하게 섞는다.
>>> myList
[[3], [2], [7], [9], [8], [1], [4], [6], [0], [5]]

>>> for i in range(3):
    print(random.randrange(0, 101, 3)) # 해당 구간에서 랜덤하게 선택

81
21
57
```

# sys 모듈

- **sys 모듈**은 파이썬 인터프리터에 대한 다양한 정보를 제공하는 모듈이다.

```
>>> import sys
```

```
>>> sys.prefix          # 파이썬이 설치된 경로  
'C:\\Users\\OWNER\\miniconda3'
```

```
>>> sys.executable
```

```
'C:\\Users\\OWNER\\miniconda3\\python.exe'
```

# time 모듈

```
>>> import time
```

```
>>> time.time() # 1970년 1월 1일 이후 지금까지 흐른 시간을 초단위로 출력  
1620096598.9858983
```



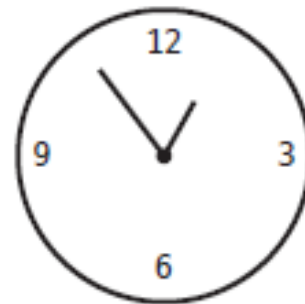
뉴욕



도쿄

1415200681

유닉스 시간



런던



# 예제

```
import time
def fib(n):      # 피보나치 수열 출력
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

start = time.time()
fib(100000)
end = time.time()
print(end-start)      # fib() 함수 실행시간 측정
```

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 ...
0.0010104179382324219
```

# calendar 모듈

```
import calendar

cal = calendar.month(2021, 5)
print(cal)
```

```
    May 2021
Mo Tu We Th Fr Sa Su
          1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

# keyword 모듈

```
import keyword

name = input("변수 이름을 입력하시오: ")

if keyword.iskeyword(name):
    print(name, "은 예약어임.")
    print("아래는 키워드의 전체 리스트임: ")
    print(keyword.kwlist)
else:
    print(name, "은 사용할 수 있는 변수이름임.")
```

변수 이름을 입력하시오: **for**  
for 은 예약어임.  
아래는 키워드의 전체 리스트임:  
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

# Lab: 동전 던지기 게임

- 동전 던지기 게임을 파이썬으로 작성해보자.
- `random` 모듈을 사용한다.

```
동전 던지기를 계속하시겠습니까?( yes, no) yes  
head  
동전 던지기를 계속하시겠습니까?( yes, no) yes  
head  
동전 던지기를 계속하시겠습니까?( yes, no) yes  
tail  
동전 던지기를 계속하시겠습니까?( yes, no) yes  
tail  
동전 던지기를 계속하시겠습니까?( yes, no) yes  
head  
동전 던지기를 계속하시겠습니까?( yes, no) no
```

# Sol:

```
import random
myList = [ "head", "tail" ]

while (True):
    response = input("동전 던지기를 계속하시겠습니까?( yes, no) ")
    if response == "yes":
        coin = random.choice(myList)
        print (coin)
    else :
        break
```

# 이번 장에서 배운 것

- 파이썬에는 어떤 객체에도 적용이 가능한 **내장 함수**가 있다. `len()`나 `max()`와 같은 함수들을 잘 사용하면 프로그래밍이 쉬워진다.
- 클래스를 정의할 때 `__iter__()`, `__next__()` 메소드를 정의하면 **이터레이터**가 된다. 이터레이터는 **for 루프**에서 사용할 수 있다.
- **연산자 오버로딩**은 `+`나 `-`와 같은 연산자들을 클래스에 맞추어서 다시 정의하는 것이다. 연산자에 해당되는 메소드(예를 들어서 `__add__(self, other)`)를 클래스 안에서 정의하면 된다.

