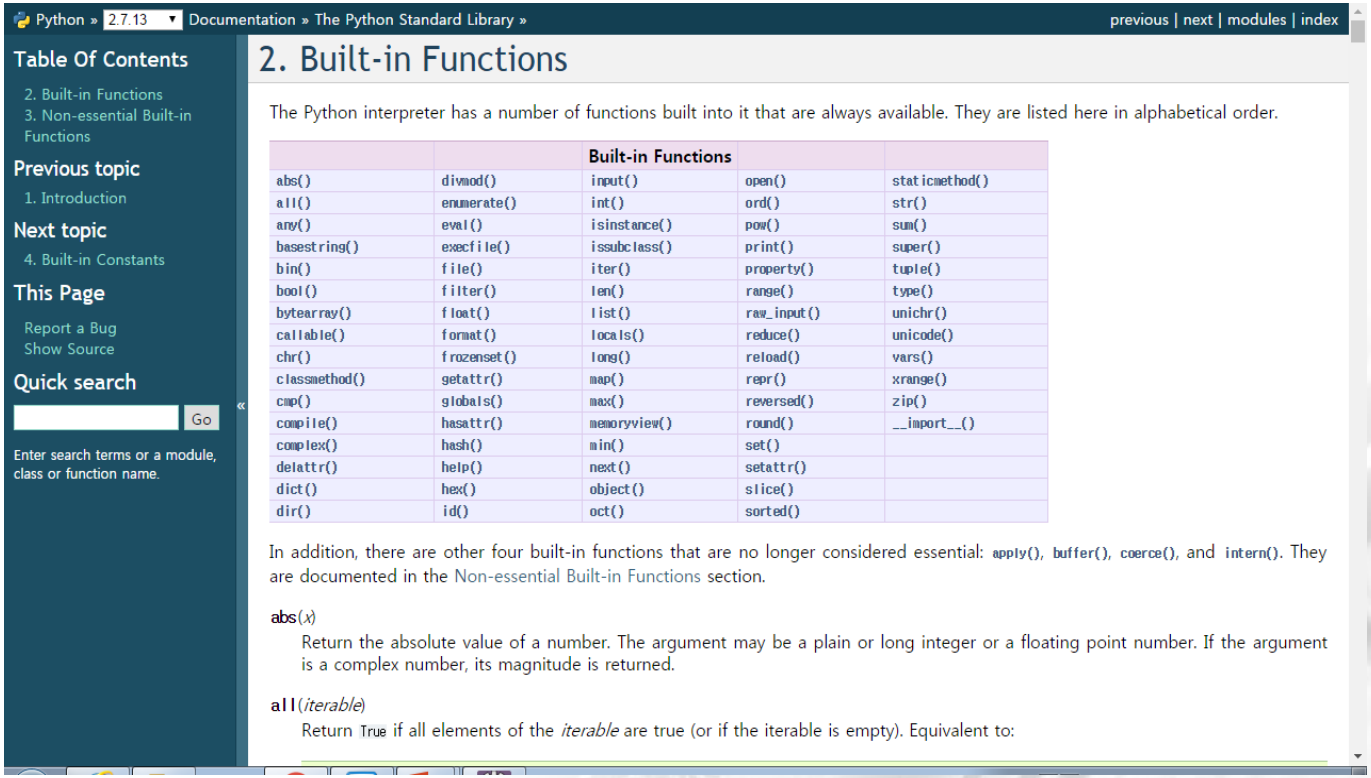
Function

Function

- 함수(Function)
  - ✓ 한 번에 실행되어야 하는 문장들을 하나의 이름으로 묶어서 독립적으로 실행할 수 있도록 하는 것
  - ✓ 함수의 이름만 호출하면 함수의 참조가 되고 함수이름()은 함수의 호출(실행)
  - ✓ 함수 사용의 장점
    - 함수 이름과 매개변수를 대입해서 호출하면 함수 안에 만들어진 코드가 실행되기 때문에 동일한 코드를 여러 번 사용해야 하는 경우 코드의 중복을 제거할 수 있어서 유지보수를 할 때 편리
    - 코드의 일정 부분을 별도의 논리적인 개념으로 분리할 수 있기 때문에 코드의 가독성을 높여 줌
  - ✓ 함수의 종류
    - Built-In Function(Maker Function, 내장 함수): 파이썬이 제공해주는 함수
    - User Define Function(사용자 정의 함수): 개발자가 만든 함수
  - ✓ 파이썬 함수는 일급 객체
    - 함수도 하나의 자료형
    - runtime에 생성하는 것이 가능
    - 변수에 대입할 수 있음
    - 함수의 매개변수로 사용 가능
    - 함수의 결과로 리턴하는 것이 가능

Function

- 내장 함수
  - ✓ 별도의 패키지나 모듈을 추가할 필요 없이 사용할 수 있는 기본적으로 제공되는 함수들
  - ✓ 내장 함수 종류: https://docs.python.org/3/library/functions.html
  - ✓ dir(__builtins__)를 이용해서도 확인 가능



Function

- 내장 함수 확인
  - ✓ max(s): 시퀀스 자료형(문자열, 리스트, 튜플)을 입력받아 그 자료가 지닌 원소 중 최대값을 리턴하는 함수
  - ✓ help(max)

```
Help on built-in function max in module builtins:

max(...)
    max(iterable, *[, default=obj, key=func]) -> value
    max(arg1, arg2, *args, *[, key=func]) -> value

    With a single iterable argument, return its biggest item. The
    default keyword-only argument specifies an object to return if
    the provided iterable is empty.
    With two or more arguments, return the largest argument.
```

Function

❖ 내장 함수 사용

```
print(max(10,30, 50))
print(max([100,300,200]))
print(max('Hello World'))
print(range(10))
print(range(3, 10))
print(range(3, 10, 2))
```

50
300
r
range(0, 10)
range(3, 10)
range(3, 10, 2)

사용자 정의 함수

- 정의 – 함수 생성
  - ✓ 파이썬에서는 함수를 정의할 때 definition(정의)를 줄인 키워드인 **def** 를 사용
  - ✓ def 다음에 함수 이름과 인수들을 나열하고 : 를 기재하고 몸체를 정의
  - ✓ 함수의 몸체는 : 다음 줄에 들여쓰기를 하고 시작해야 하며 파이썬은 어떤 형식의 데이터도 변수에 대입할 수 있기 때문에 인수의 자료형은 기재하지 않아도 됨 – 기재하고자 할 때는 : 을 추가하고 자료형을 기재
  - ✓ return 은 함수의 수행을 종료하고 작업한 결과를 함수를 호출한 곳으로 돌려줄 때 사용
  - ✓ def 키워드를 이용한 함수 정의

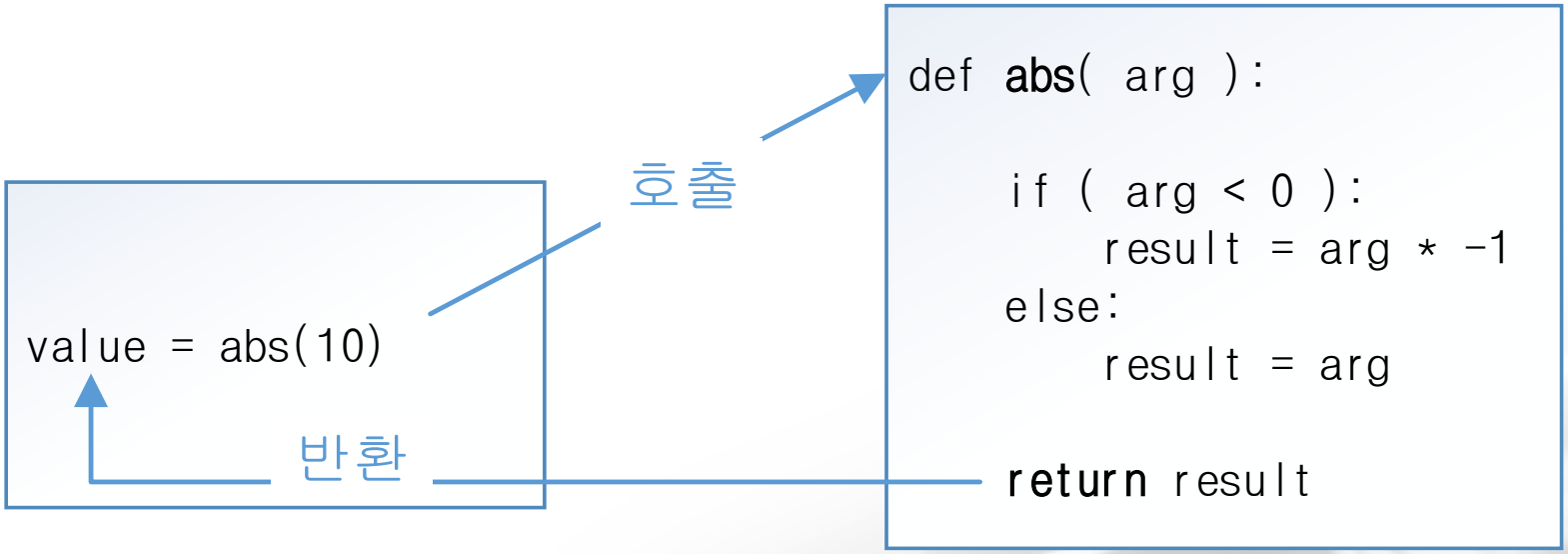
```
def 함수이름(인수들):
  문장을 나열
  return <값>
```

사용자 정의 함수

- 호출(call)
  - ✓ 함수 이름만 사용하면 함수의 참조를 리턴
  - ✓ 함수 이름(매개변수)의 형태로 작성하면 파이썬은 함수 이름 내부에 정의되어 있는 코드를 별도의 메모리를 할당 받아서 수행하고 종료하면 호출한 곳으로 돌아감

사용자 정의 함수

- 리턴(return)
  - ✓ 함수가 자신의 코드를 실행하고 호출하는 곳으로 돌아가는 제어문
  - ✓ 이 때 함수가 수행한 결과를 자신의 이름을 호출한 코드에게 돌려주고자 하는 경우 return 이라는 명령어 뒤에 리턴할 데이터를 기재



사용자 정의 함수

- 리턴(return)
  - ✓ return 문장 다음에 값이 없으면 None을 리턴
  - ✓ 파이썬은 리턴되는 자료형을 작성하지 않아도 되지만 def 함수명(매개변수 나열) -> 리턴타입 의 형태로 작성할 수 있고 리턴되는 데이터가 없을 때는 None 이라고 기재하는 것이 가능
  - ✓ return을 할 때 여러 개의 데이터를 return 하려면 튜플이나 리스트 또는 클래스의 객체 등을 만들어서 리턴
  - ✓ 여러 개의 데이터를 ,로 구분해서 리턴하면 튜플로 리턴됨

```
def nothing():
    return

print(nothing())

def swap(a, b):
    return b, a

a = 10
b = 20
print(swap(a,b))
```

=> 첫 번째 출력문은 nothing 함수에서 return 뒤에 데이터가 없으므로 None

=> 두 번째 출력문은 swap 함수에서 2개의 값을 리턴 했으므로 튜플로 만들어져서 (20, 10)

사용자 정의 함수

- Argument(매개변수, Parameter)
  - ✓ 함수를 호출하는 곳에서 함수에게 전달하는 데이터
  - ✓ 매개변수 이름만 작성해도 되지만 이름:자료형 의 형태로 작성해도 됨
  - ✓ 매개변수가 있는 함수를 호출할 때 매개변수를 대입하지 않으면 에러 발생
  - ✓ 파이썬에서는 매개변수(argument)에 데이터를 넘겨주면 Scala(1개인 데이터) Data는 변수 이름이 데이터 자체를 의미하므로 값이 넘어가고 Vector(0개 이상의 데이터) Data는 변수가 데이터의 시작 위치(참조)를 의미하므로 값의 시작 위치를 넘김

사용자 정의 함수

- Argument(매개변수, Parameter)
  - ✓ Call By Value

```
def f(t):
    t = 10
    print("t:", t)


#a가 저장하고 있는 20이라는 데이터를 넘겼으므로 함수를 호출해도 a가 가리키고 있는
    데이터는 변경되지 않음
a = 20
f(a)
print("a:", a)
```

사용자 정의 함수

- Argument(매개변수, Parameter)
  - ✓ Call By Reference

```
def f(t):
    t[0] = 100

li = [1,2,3]

f(li)

print(li[0])
```

=>list가 저장하고 있는 참조를 넘겨주었기 때문에 함수 내에서 리스트의 내용을 변경하면 원본의 데이터도 변경됨

=>따라서 li[0]의 값은 100

사용자 정의 함수

- ❖ 매개변수
  - ✓ 매개변수의 기본값
    - 함수에 매개변수가 있는 경우 매개변수를 넘겨주지 않으면 에러가 발생
    - 함수의 매개변수에 기본값을 할당할 수 있는데 매개변수이름=값 의 형식을 이용해서 매개변수를 정의하면 함수를 호출할 때 매개변수에 값을 대입하지 않는 경우 기본값을 사용하고 호출할 때 값을 대입하게 되면 대입한 값을 사용
    - 기본값이 있는 매개변수 뒤에는 기본값이 없는 매개변수가 올 수 없음
  - ✓ 함수를 호출할 때 매개변수의 이름과 함께 값을 대입해서 호출할 수 있는데 이렇게 되면 매개변수의 순서를 변경해서 대입한 후 호출하는 것이 가능

사용자 정의 함수

- 매개변수
  - ✓ 매개변수의 기본값 사용

```
def f(n, step=1):
    return n+step;

a = 1
#순서대로 값을 대입해서 호출
result = f(a,10)
print(result)

b = 1
#첫번째 매개변수에만 값을 대입해서 호출
result = f(b)
print(result)

#매개변수 이름을 이용해서 매개변수를 대입해서 호출
result = f(step=2, n=10)
print(result)
```

```
11
2
12
```

사용자 정의 함수

- 매개변수
  - ✓ sum 함수의 원형

sum(iterable, start=0, /)
Iterable은 데이터의 모임이며 start는 처음 시작할 때 가지는 기본값

```
#sum 함수의 도움말 확인 과 사용
help(sum)
print(sum([100, 200, 300], 2))

#두번째 매개변수를 생략해서 실행
print(sum([100, 200, 300]))
```

602
600

사용자 정의 함수

- 매개변수
  - ✓ 순수 함수(pure function)
    - 함수의 실행이 외부 상태에 영향을 끼치지 않는 함수
    - 순수 함수는 side effect 가 없어야 하고 입력 값이 같으면 언제나 동일한 출력 값을 반환
  - ✓ 비순수 함수(impure function)
    - modifier function 라고도 하는데 함수의 실행이 외부 상태에 영향을 끼치는 함수

사용자 정의 함수

- 매개변수의 unpacking
  - ✓ 매개변수로 list 나 tuple 또는 set을 대입할 때 * 과 함께 대입하면 자동으로 압축이 해제되서 순서대로 매개변수에 대입됨
  - ✓ dict 는 key가 대입되며 ** 을 이용해서 대입하면 매개변수 이름 과 대응되는 key 이름으로 매칭

```
#unpacking
def unpacking(first, second):
    print('첫번째 데이터:', first)
    print('두번째 데이터:', second)

unpacking(*[100, 200])
print()
unpacking(*(100, 200))
print()
unpacking(*{100, 200})
print()
unpacking(*{"a":100, "b":200})
print()
```

```
첫번째 데이터: 100
두번째 데이터: 200

첫번째 데이터: 100
두번째 데이터: 200

첫번째 데이터: 200
두번째 데이터: 100

첫번째 데이터: a
두번째 데이터: b
```

사용자 정의 함수

❖ 매개변수의 unpacking

```
#unpacking
def personal_info(name, age, gender):
    print('이름:', name)
    print('나이:', age)
    print('성별:', gender)

person = {"name":"아담", "age":25, "gender":"남자"}
personal_info(**person)
```

```
이름: 아담
나이: 25
성별: 남자
```

사용자 정의 함수

- 가변 매개변수(Arbitrary Argument List)
  - ✓ 입력 개수가 달라질 수 있는 매개변수
  - ✓ *를 이용하여 정의된 가변 매개변수는 튜플이고 **을 이용하여 정의된 매개변수는 dict

```
def 함수이름(*매개변수):
    코드블록
```

매개변수 앞에 *를 붙이면 해당매개변수는 가변으로 지정

사용자 정의 함수

- 가변 매개변수(Arbitrary Argument List)

```
# 가변 매개변수
def merge_string(*text_list):
    print('text_list:', type(text_list))
    result = ''
    for s in text_list:
        result = result + " " + s
    return result
print(merge_string('안녕하세요', '반갑습니다'))
print(merge_string('오늘은', '날씨가', '매우 춥습니다.'))
```

```
text_list: <class 'tuple'>
 안녕하세요 반갑습니다
text_list: <class 'tuple'>
 오늘은 날씨가 매우 춥습니다.
```

사용자 정의 함수

- 일반 매개변수와 함께 사용하는 가변매개변수

```
def print_args(argc, *argv):
        for i in range(argc):
                print(argv[i])

print_args(3, "argv1", "argv2", "argv3")

print_args(argc=3, "argv1", "argv2", "argv3")
```

SyntaxError: non-keyword arg after keyword arg

가변 매개변수 앞에 정의된 일반 매개변수는 키워드 매개변수로 호출할 수 없음

사용자 정의 함수

❖ 가변 매개변수와 함께 사용하는 일반 매개변수

```
def print_args(*argv, argc):
        for i in range(argc):
                print(argv[i])

print_args("argv1", "argv2", "argv3", argc=3)
print_args("argv1", "argv2", "argv3", 3)
```

가변 매개변수 뒤에 정의된 일반 매개변수는 반드시 키워드 매개변수로 호출

```
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    print_args("argv1", "argv2", "argv3", 3)
TypeError: print_args() missing 1 required keyword-only argument: 'argc'
```

사용자 정의 함수

- 정의되지 않은 매개변수 전달
  - ✓ 매개변수의 이름 앞에 **을 붙이면 매개변수를 dict 형태로 전달
  - ✓ 매개변수 이름과 값을 대입하면 dict로 변환해서 대입
  - ✓ 이 매개변수는 매개변수의 마지막에 위치해야 함
  - ✓ 정의되지 않은 매개변수 전달

```
#정의되지 않은 매개변수
def userURIBuilder(server, port, **param):
    print('param:', type(param))
    uri = "http://" + server + ":" + "/"
    for attr in param:
        uri = uri + attr + "=" + param[attr] + "&"
    return uri

print(userURIBuilder("test.com", "8080", id="cyberadam", pw="metaverse"))
print(userURIBuilder("test.com", "8080", id="cyberadam", pw="metaverse", nick="군계"))
```

```
param: <class 'dict'>
http://test.com:/id=cyberadam&pw=metaverse&;
param: <class 'dict'>
http://test.com:/id=cyberadam&pw=metaverse&nick=군계&;
```

사용자 정의 함수

- 재귀 함수(Recursive Function)
  - ✓ 함수가 자기 자신을 다시 호출하는 것을 재귀호출(Recursive Call)이라 함.
  - ✓ 재귀 함수의 예

```
#재귀 구현
def some_func(count):
    if count > 0:
        some_func(count-1)
    else:
        return
    print(count)

some_func(5)
```

```
1
2
3
4
5
```

사용자 정의 함수

- 재귀 함수
  - ✓ 합계

```
def tot(n):
    if n == 1:
        return 1
    elif n > 1:
        return tot(n - 1) + n

print(tot(5))

print(tot(10))

print(tot(100))
```

```
15
55
5050
```

사용자 정의 함수

- 재귀 함수
  - ✓ 팩토리얼

```
def factorial(n):
    if n == 0:
        return 1
    elif n > 0:
        return factorial(n - 1) * n

print(factorial(5))

print(factorial(10))

print(factorial(100))
```

```
120
3628800
933262154439441526816992388562667004907159682643816214685929638952175999
    932299156089414639761565182862536979208272237582511852109168640000000
    0000000000000000
```

사용자 정의 함수

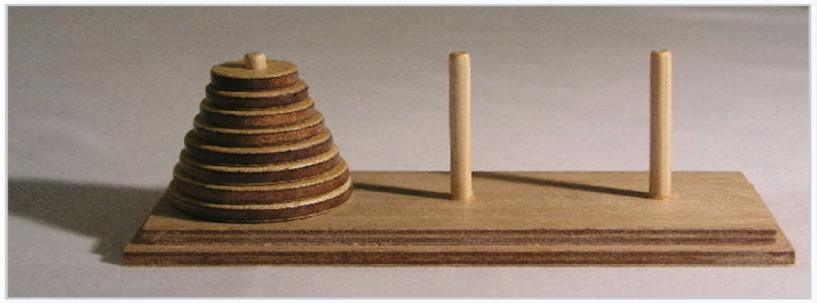
- 재귀 함수
  - ✓ 피보나치 수열
    - (1,1,2,3,5,8,13…)
    - 첫번째와 두번째 항은 1 나머지 항은 이전 2개 항의 합

```
#피보나치 수열
def fibonacci(n):
        if n == 1 or n == 2:
                return 1
        else:
                return fibonacci(n-1) + fibonacci(n-2)
print(fibonacci(5))
print(fibonacci(7))
print(fibonacci(10))
```

```
5
13
55
```

사용자 정의 함수

- 재귀 함수
  - ✓ 하노이의 탑
    - 하노이의 탑(Tower of Hanoi)은 퍼즐의 일종
    - 세 개의 기둥과 이 기둥에 꽂을 수 있는 크기가 다양한 원판들이 있고 퍼즐을 시작하기 전에는 한 기둥에 원판들이 작은 것이 위에 있도록 순서대로 쌓여 있음
    - 게임의 목적은 다음 두 가지 조건을 만족시키면서 한 기둥에 꽂힌 원판들을 그 순서 그대로 다른 기둥으로 옮겨서 다시 쌓는 것
      - 한 번에 한개의 원판만 옮길 수 있음
      - 큰 원판이 작은 원판 위에 있어서는 안됨
    - 하노이의 탑 문제는 재귀 호출을 이용하여 풀 수 있는 가장 유명한 예제 중의 하나
    - 프로그래밍에서 알고리즘 예제로 많이 사용
    - 일반적으로 원판이 n개 일 때, 2^n -1번의 이동으로 원판을 모두 옮길 수 있음(2^n - 1는 메르센 수라고 부름)



하노이의 탑

사용자 정의 함수

- 재귀 함수
  - ✓ 하노이의 탑

```
def hanoi(ndisks, startPeg=1, endPeg=3):
    if ndisks:
        hanoi(ndisks-1, startPeg, 6-startPeg-endPeg)
        print(startPeg, "번 기둥의", ndisks, "번 고리를 ", endPeg, "번 기둥에 옮깁니다.")
        hanoi(ndisks-1, 6-startPeg-endPeg, endPeg )

hanoi(ndisks=3)
```

```
1 번 기둥의 1 번 고리를  3 번 기둥에 옮깁니다.
1 번 기둥의 2 번 고리를  2 번 기둥에 옮깁니다.
3 번 기둥의 1 번 고리를  2 번 기둥에 옮깁니다.
1 번 기둥의 3 번 고리를  3 번 기둥에 옮깁니다.
2 번 기둥의 1 번 고리를  1 번 기둥에 옮깁니다.
2 번 기둥의 2 번 고리를  3 번 기둥에 옮깁니다.
1 번 기둥의 1 번 고리를  3 번 기둥에 옮깁니다.
```

사용자 정의 함수

- pass
  - ✓ 함수 나 클래스가 아무런 동작도 수행하지 않아야 하는 경우에 사용
  - ✓ 일단 이름만 만들어두고 나중에 내용을 작성해야 사용할 목적으로 사용

```
def temp():
    pass
```

사용자 정의 함수

- 매개변수의 자료형 과 리턴 타입 기재
  - ✓ 기존 방식

```
def isPalindrome(s):
    strs = []
    for char in s:
        if char.isalnum():
            strs.append(char.lower())

    # 팰린드롬 여부 판별
    while len(strs) > 1:
        if strs.pop(0) != strs.pop():
            return False

    return True

print(isPalindrome('eros'))
print(isPalindrome('역삼역'))
print(isPalindrome('Ada'))
```

사용자 정의 함수

- 매개변수의 자료형 과 리턴 타입 기재
  - ✓ 자료형 명시

```
def isPalindrome(s: str) -> bool:
    strs = []
    for char in s:
        if char.isalnum():
            strs.append(char.lower())

    # 팰린드롬 여부 판별
    while len(strs) > 1:
        if strs.pop(0) != strs.pop():
            return False

    return True

print(isPalindrome('eros'))
print(isPalindrome('역삼역'))
print(isPalindrome('Ada'))
```

사용자 정의 함수

- 함수 선언에 애너테이션 추가
  - ✓ 함수 선언에서 각 매개변수에는 콜론(:) 뒤에 애너테이션 표현식을 추가할 수 있음
  - ✓ 기본값이 있을 때, 애너테이션은 인수명과 등호(=) 사이
  - ✓ 반환값에 애너테이션을 추가하려면 매개변수를 닫는 괄호와 함수 선언의 제일 뒤에 오는 콜론 사이에 -> 기호와 표현식을 추가
  - ✓ 표현식은 어떤 자료형도 될 수 있음

```
#함수에 애너테이션 추가
def clip(text:str, max_len: 'int > 0' = 80) -> None:
    print(text)
    print(max_len)

clip("Annotation", 100)
help(clip)
```

```
Annotation
100
Help on function clip in module __main__:

clip(text: str, max_len: 'int > 0' = 80) -> None
    #함수에 애너테이션 추가
```

함수 객체의 속성

- 함수의 도움말
  - ✓ 파이썬에서는 help 함수를 이용해서 함수의 도움말을 확인
  - ✓ 함수에 도움말을 추가하고자 하는 경우
    - 함수를 만들고 함수의 __doc__ 속성에 도움말로 표시될 부분을 추가
    - 함수를 정의할 때 가장 상단에 "" 또는 """ """ 사이에 문자열 삽입

함수 객체의 속성

- 함수의 도움말
  - ✓ 도움말 작성

```
def plus(a, b):
    return a+b
plus.__doc__ = "2개의 숫자 데이터를 받아서 덧셈한 결과를 리턴해주는 함수"
def minus(a,b):
    """
    2개의 숫자 데이터를 받아서 뺄셈한 결과를
    리턴해주는 함수
    """
help(plus)
help(minus)
```

```
Help on function plus in module __main__:
plus(a, b)
    2개의 숫자 데이터를 받아서 덧셈한 결과를 리턴해주는 함수
Help on function minus in module __main__:
minus(a, b)
    2개의 숫자 데이터를 받아서 뺄셈한 결과를
    리턴해주는 함수
```

함수 객체의 속성

- 함수 객체의 속성
  - ✓ __name__: 함수의 이름
  - ✓ __defaults__: 기본 인수 값
  - ✓ __code__
    - 코드 객체
    - 의사 컴파일 된(Pseudo-Compiled) 실행 가능한 파이썬 코드
    - compile() 내장 함수에 의해 반환되고 함수 객체의 __code__ 속성으로 참조
  - ✓ __globals__: 함수의 전역 영역

함수 객체의 속성

- 함수 객체의 속성

```
def f(a, b, c = 1):
    localx = 1
    localy = 2
    return 1

print(f.__name__)
print()
print(f.__defaults__)
print()
print(f.__code__)
print()
print(f.__globals__)
```

함수형 프로그래밍

- 함수를 변수에 저장
  - ✓ 함수 이름만 사용하면 함수가 저장된 곳의 참조를 의미
  - ✓ 일반적으로 함수를 매개변수로 전달하고자 하는 경우에 이 방법을 사용

```
#함수를 변수에 저장
def print_something(a):
        print(a)

p = print_something
p(123)
```

```
123
```

함수형 프로그래밍

- 고위 함수(higher-order function)
  - ✓ 함수를 인수로 받거나, 함수를 결과로 반환하는 함수

```
#함수를 매개변수로 사용
def plus(a, b):
        return a+b

def minus(a, b):
        return a-b

def cal(func, a, b):
        return func(a, b)

print(cal(plus, 1, 2))
print(cal(minus, 1, 2))
```

```
3
-1
```

함수형 프로그래밍

- 고위 함수(higher-order function)
  - ✓ 함수를 리턴

```
def hello_korean():
        print('안녕하세요.')

def hello_english():
        print('Hello.')

def get_greeting(where):
        if where == 'K':
            return hello_korean
        else:
            return hello_english

hello = get_greeting('K')
hello()

hello = get_greeting('E')
hello()
```

안녕하세요.
Hello.

함수형 프로그래밍

- lambda
  - ✓ 이름이 없는 한 줄 짜리 함수
  - ✓ 작성 방법

    lambda <인수 나열>:<리턴할 내용>
  - ✓ 인수가 없으면 생략 가능

    항상 1을 리턴하는 람다 함수

    f = lambda:1

    2개의 인수를 받아서 합을 구해주는 람다 함수

    f = lambda x , y : x+y
  - ✓ 람다 함수도 매개변수의 기본값을 가질 수 있으며 외부 변수 사용 가능
  - ✓ 람다 안에서는 새로운 변수 선언은 안됨
  - ✓ 할당문이나 while, try 등의 파이썬 문장을 사용할 수 없음

함수형 프로그래밍

- lambda

```
def g(func):
    return [func(x) for x in range(-10, 10)]

print(g(lambda x:x * x + 3 * x -10))

print(g(lambda x:x*x*x))
y = 10
print(g(lambda x: x + y))
```

[60, 44, 30, 18, 8, 0, -6, -10, -12, -12, -10, -6, 0, 8, 18, 30, 44, 60, 78, 98]
[-1000, -729, -512, -343, -216, -125, -64, -27, -8, -1, 0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

함수형 프로그래밍

- map 내장 함수
  - ✓ 컬렉션과 함수를 매개변수로 받아서 컬렉션의 모든 데이터를 함수의 매개변수로 대입해서 결과를 iterator 인스턴스로 리턴하는 함수
  - ✓ 대입되는 함수는 하나의 매개변수를 받아서 하나의 값을 리턴하는 함수이어야 함

함수형 프로그래밍

❖ map 내장 함수

```
import datetime

def f(x):
    return x*x

li = [i for i in range(10000)]

print('반복문을 이용한 실행')
s1 = datetime.datetime.now()

for imsi in li:
    print(f(imsi), end=" ")
s2 = datetime.datetime.now()

print('₩nmap을 이용한 실행')
s3 = datetime.datetime.now()
result = list(map(f,li))
s4 = datetime.datetime.now()

print("₩n반복문 작업 시작 시간:", s1)
print("반복문 작업 종료 시간:", s2)
print("₩nmap 작업 시작 시간:", s3)
print("map 작업 종료 시간:", s4)
```

```
반복문 작업 시작 시간: 2021-01-15 17:22:45.338837
반복문 작업 종료 시간: 2021-01-15 17:22:46.911658
map 작업 시작 시간: 2021-01-15 17:22:46.911758
map 작업 종료 시간: 2021-01-15 17:22:46.912736
```

함수형 프로그래밍

- filter 내장 함수
  - ✓ 컬렉션과 리턴 타입이 bool 인 함수를 매개변수로 받아서 컬렉션의 모든 데이터를 함수의 매개변수로 대입해서 리턴되는 결과가 True인 경우만 iterator 인스턴스로 리턴하는 함수
  - ✓ 대입되는 함수는 하나의 매개변수를 받아서 bool 을 리턴하는 함수 이어야 함

```
def odd(x):
    return x % 2 == 1

li = [1,2,3,4,5]
print('filter를 이용한 홀수 골라내기')
result = list(filter(odd, li))
print(result)
```

```
filter를 이용한 홀수 골라내기
[1, 3, 5]
```

함수형 프로그래밍

❖ filter 내장 함수

```
li = [1,2,3,4,5]
print('lambda와 filter을 이용한 실행')
result = list(filter(lambda x : x%2==0, li))
print(result)
```

```
lambda와 map을 이용한 실행
[1, 4, 9, 16, 25]
```

함수형 프로그래밍

- reduce 내장 함수
  - ✓ python 3에서 내장 함수에서 제외
  - ✓ functools 패키지의 함수로 변경됨
  - ✓ 처음 데이터 2개의 값을 매개변수로 대입된 함수를 실행해서 결과를 만든 후 그 결과와 다음 데이터를 매개변수로 해서 연산을 수행하는 함수

```
from functools import reduce
result = reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])
print(result)
```

15

함수형 프로그래밍

❖ reduce 내장 함수

```
from functools import reduce

li = [70, 80, 98, 77, 95]
#전통적으로 최대값 구하기
def maximum(li):
    default = 0
    for e in li:
        if default < e:
            default = e
    return default
print(maximum(li))

#reduce 활용하여 최대값 구하기
print(reduce(lambda a,b: a if a > b else b ,li))
```

```
98
98
```

함수형 프로그래밍

- zip 내장 함수
  - ✓ 길이가 같은 반복 가능한 데이터(목록, 사전 등)를 하나로 묶어 반환하는 함수
  - ✓ 형태가 다른 값들을 하나의 구조나 변수로 활용하기 위해 사용
  - ✓ 결괏값 = zip(범위 1, 범위 2, ...)의 형태로 사용
  - ✓ 목록(list), 집합(set), 사전(dict) 등을 묶어 결과를 반환

```
a = "Adam"
b = [1, 2, 3, 4]
c = ("하나", "둘", "셋", "넷")

print(list(zip(a, b, c)))
print(set(zip(a, b, c)))
print(dict(zip(a, b)))
```

```
[('A', 1, '하나'), ('d', 2, '둘'), ('a', 3, '셋'), ('m', 4, '넷')]
{('d', 2, '둘'), ('A', 1, '하나'), ('a', 3, '셋'), ('m', 4, '넷')}
{'A': 1, 'd': 2, 'a': 3, 'm': 4}
```

함수형 프로그래밍

- zip 내장 함수
  - ✓ 반복문 적용

```
L1 = ["A", "B", "C", "D"]
L2 = ["가", "나", "다", "라"]


for i, j in zip(L1, L2):
    print(i, j)
```

```
A 가
B 나
C 다
D 라
```

함수형 프로그래밍

- 중첩 함수(Nested Function)
  - ✓ 함수 안에 정의된 함수
  - ✓ 중첩 함수는 자신이 소속되어 있는 함수의 매개변수에 접근할 수 있음
  - ✓ 중첩 함수는 자신이 소속되어 있는 함수 외부에서는 보이지 않음

```
import math
#외부 함수
def stddev(*args):
    #내부 함수 - 평균
     def mean():
              return sum(args)/len(args)
    #내부 함수 - 분산
     def variance(m):
              total = 0
              for arg in args:
                       total += (arg - m ) ** 2
              return total/(len(args)-1)
     v = variance(mean())
    #표준편차를 리턴하는 함수
     return math.sqrt(v)

print(stddev(2.3, 1.7, 1.4, 0.7, 1.9))
```

0.6

함수형 프로그래밍

❖ 중첩 함수(Nested Function) :

```
mean()

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/var/folders/42/fxl3_n31121f16yj69fbrb_m0000gn/T/ipykernel_8664/1856946037.py in
<module>
----> 1 mean()

NameError: name 'mean' is not defined
```

함수형 프로그래밍

- 유효 범위
  - ✓ 함수 밖에서 변수 a를 정의하여 0을 대입하고 함수 안에서 변수 a를 또 정의하여 1을 대입한 경우 이 함수를 실행(호출)하고 나면 함수 밖에서 정의된 변수 a의 값은?
    - 답 : 0
    - 함수 밖에 있는 a와 안에 있는 a는 이름은 같지만 사실은 완전히 별개의 변수
  - ✓ 실습 1

```
def scope_test():
        #지역 변수 a 생성
        a = 1
        print('a : {0}'.format(a))

a = 0
scope_test()
print('a : {0}'.format(a))
```

```
a : 1
a : 0
```

함수형 프로그래밍

- 유효 범위
  - ✓ 변수는 자신이 생성된 범위(코드블록) 안에서만 유효
  - ✓ 함수 안에서 만든 변수는 함수 안에서만 살아있다가 함수 코드의 실행이 종료되면 그 생명이 다해서 소멸되는데 이것을 지역변수(Local Variable)라고 함
  - ✓ 이와는 반대로 함수 외부에서 만든 변수는 프로그램이 살아있는 동안에는 함께 살아있다가 프로그램이 종료될 때 같이 소멸됨
  - ✓ 파이썬은 함수 안에서 생성되는 모든 변수를 지역 변수로 간주
  - ✓ 함수 내부에서 외부 변수를 사용하기 위해서는 **global** 키워드를 이용
  - ✓ 함수 내에서 자신을 포함하는 함수의 변수를 사용할 때는 **nonlocal** 키워드를 사용
  - ✓ global 이나 nonlocal 변수를 사용할 때는 선언을 먼저 하고 사용

함수형 프로그래밍

- 변수의 유효범위

```
def outer():
        a = 1
        def inner():
            nonlocal a
            print("함수의 외부 함수에 있는 a:{0}".format(a))
            a = 10
        inner()
        print("내부함수에서 변경한 경우의 a:{0}".format(a))

a = 0
outer()
print("a:{0}".format(a))
```

```
함수의 외부 함수에 있는 a:1
내부함수에서 변경한 경우의 a:10
a:0
```

함수형 프로그래밍

- 변수의 유효범위

```
def outer():
        a = 1
        def inner():
            global a
            print("함수의 외부에 있는 a:{0}".format(a))
            a = 10
        inner()

a = 0
outer()
print("a:{0}".format(a))
```

```
함수의 외부에 있는 a:0
a:10
```

함수형 프로그래밍

- Closure
  - ✓ 함수 내에서 함수를 리턴하는 형태를 만들어서 지역 변수의 값을 계속 사용하기 위한 개념

```
def calc():
    a = 3
    b = 5
    def mul_add(x):
        return a * x + b    # 함수 바깥의 지역 변수 a, b를 사용하여 계산
    return mul_add          # mul_add 함수를 반환

c = calc()
print(c(1), c(2))
```

```
8 11
```

함수형 프로그래밍

- Closure

```
def calc():
    a = 3
    b = 5
    return lambda x: a * x + b     # 람다 표현식을 반환

c = calc()
print(c(1), c(2))
```

8 11

함수형 프로그래밍

- 일급 객체
  - ✓ 일급 객체(firs&lass object)란 다음 조건을 만족하는 객체
    - 변수 나 데이터 구조에 넣을 수. 있어야 함
    - 매개변수에 전달할 수 있어야 함
    - 반환값으로 사용할 수 있어야 함
  - ✓ 일급 함수(first-class function)는 일급 객체의 조건을 만족하면서 실행 중(run-time)에 함수를 생성할 수 있어야 함
  - ✓ 파이썬에서는 def 안에서 def로 함수를 만들거나 lambda를 사용하여 실행 중에 함수를 생성할 수 있으므로 파이썬의 함수는 일급 함수

함수형 프로그래밍

- 일급 객체
  - ✓ 일급 객체(firs&lass object)란 다음 조건을 만족하는 객체
    - 변수 나 데이터 구조에 넣을 수. 있어야 함
    - 매개변수에 전달할 수 있어야 함
    - 반환값으로 사용할 수 있어야 함
  - ✓ 일급 함수(first-class function)는 일급 객체의 조건을 만족하면서 실행 중(run-time)에 함수를 생성할 수 있어야 함
  - ✓ 파이썬에서는 def 안에서 def로 함수를 만들거나 lambda를 사용하여 실행 중에 함수를 생성할 수 있으므로 파이썬의 함수는 일급 함수

Decorator

- Decorator
  - ✓ 다른 함수를 인수로 받아서 어떤 처리를 수행하고 함수를 리턴하거나 함수를 다른 함수나 객체로 대체하는 것
  - ✓ 데코레이터는 함수를 수정하지 않은 상태에서 추가 기능을 구현할 때 사용
  - ✓ 함수의 시작과 끝을 출력하고 싶다면 다음과 같이 함수 시작, 끝 부분에 print를 넣어주어야 하는데 데코레이터를 이용하면 이럴 필요가 없음
  - ✓ 예를 들어 decorate 라는 이름의 Decorator 가 존재하는 경우

```
@decorate
def target():
    print('running target()')
```

위와 같은 코드가 존재하는 경우 아래처럼 동작

```
def target():
    print('running target()')
target = decorate(target)
```

Decorator

- Decorator
  - ✓ Decorator 사용

```
#Decorator 사용
def deco(func):
    def inner():
        print('running inner()')
    return inner #deco () 가 inner () 함수 객체를 반환
@deco
def target(): #target ( )을 deco로 Decorate
    print('running target()')

target() #Decorate된 target()을 호출하면 실제로는 inner ()를 실행
```

running inner()

Decorator

- Decorator
  - ✓ 로깅을 위한 Decorate

```
import time

#Decorate 된 함수를 호출할 때 마다 시간을 측정해서 실행에 소요된 시간, 전달된 인수,
    반환값을 출력하는 Decorate
def clock(func):
    def clocked(*args):
        t0 = time.time()
        result = func(*args)
        elapsed = time.time() - t0
        name = func.__name__
        arg_str = ', '.join(repr(arg) for arg in args)
        print('[%0.8fs] %s(%s) -> %r' % (elapsed, name, arg_str, result))
        return result
    return clocked
```

Decorator

- Decorator
  - ✓ 로깅을 위한 Decorate

```
@clock
def snooze(seconds):
    time.sleep(seconds)

@clock
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)

print('*' * 40, 'Calling snooze(.123)')
snooze(.123)
print('*' * 40, 'Calling factorial(6)')
print('6! =', factorial(6))
```

Decorator

- Decorator
  - ✓ 로깅을 위한 Decorate

```
**************************************** Calling snooze(.123)
[0.12683082s] snooze(0.123) -> None
**************************************** Calling factorial(6)
[0.00000000s] factorial(1) -> 1
[0.00001383s] factorial(2) -> 2
[0.00002098s] factorial(3) -> 6
[0.00002789s] factorial(4) -> 24
[0.00003624s] factorial(5) -> 120
[0.00004482s] factorial(6) -> 720
6! = 720
```

Decorator

- Decorator
  - ✓ 표준 라이브러리에서 제공하는 functools.lru_cache()를이용한 메모이제이션
    - 메모이제이션은 이전에 실행한 함수의 결과를 저장함으로써 이전에 사용된 인수에 대해 다시 계산할 필요가 없게 해주는 것
    - LRU는 Least Recently Used 약자로서 오랫동안 사용하지 않은 항목을 버림으로써 캐시가 무한정 커지지 않음을 의미

Decorator

- Decorator
  - ✓ 느리게 동작하는 피보나치 함수

```
import time
def clock(func):
    def clocked(*args):
        t0 = time.time()
        result = func(*args)
        elapsed = time.time() - t0
        name = func.__name__
        arg_str = ', '.join(repr(arg) for arg in args)
        print('[%0.8fs] %s(%s) -> %r' % (elapsed, name, arg_str, result))
        return result
    return clocked

@clock
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-2) + fibonacci(n-1)

print(fibonacci(6))
```

Decorator

- Decorator
  - ✓ 느리게 동작하는 피보나치 함수

```
[0.00000000s] fibonacci(0) -> 0
[0.00000072s] fibonacci(1) -> 1
[0.00008798s] fibonacci(2) -> 1
[0.00000095s] fibonacci(1) -> 1
[0.00000000s] fibonacci(0) -> 0
[0.00000119s] fibonacci(1) -> 1
[0.00002527s] fibonacci(2) -> 1
[0.00004816s] fibonacci(3) -> 2
[0.00015998s] fibonacci(4) -> 3
[0.00000000s] fibonacci(1) -> 1
[0.00000000s] fibonacci(0) -> 0
[0.00000072s] fibonacci(1) -> 1
[0.00002098s] fibonacci(2) -> 1
[0.00004101s] fibonacci(3) -> 2
[0.00000119s] fibonacci(0) -> 0
[0.00000095s] fibonacci(1) -> 1
[0.00001693s] fibonacci(2) -> 1
```

Decorator

- Decorator
  - ✓ 느리게 동작하는 피보나치 함수

```
[0.00000000s] fibonacci(1) -> 1
[0.00000095s] fibonacci(0) -> 0
[0.00000095s] fibonacci(1) -> 1
[0.00001478s] fibonacci(2) -> 1
[0.00002718s] fibonacci(3) -> 2
[0.00005984s] fibonacci(4) -> 3
[0.00011706s] fibonacci(5) -> 5
[0.00029802s] fibonacci(6) -> 8
8
```

Decorator

- Decorator
  - ✓ 메모이제이션을 이용한 피보나치 함수

```
import functools
import time
def clock(func):
    def clocked(*args):
        t0 = time.time()
        result = func(*args)
        elapsed = time.time() - t0
        name = func.__name__
        arg_str = ', '.join(repr(arg) for arg in args)
        print('[%0.8fs] %s(%s) -> %r' % (elapsed, name, arg_str, result))
        return result
    return clocked

@functools.lru_cache()
@clock
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-2) + fibonacci(n-1)

print(fibonacci(6))
```

Decorator

- Decorator
  - ✓ 메모이제이션을 이용한 피보나치 함수

```
[0.00000095s] fibonacci(0) -> 0
[0.00000000s] fibonacci(1) -> 1
[0.00006795s] fibonacci(2) -> 1
[0.00000119s] fibonacci(3) -> 2
[0.00008392s] fibonacci(4) -> 3
[0.00000000s] fibonacci(5) -> 5
[0.00010085s] fibonacci(6) -> 8
8
```

연습 문제

- N 번째 피보나치 수열의 값을 리턴해주는 함수를 재귀를 이용하지 않는 경우 와 재귀를 이용하는 형태 2가지로 구현해보고 50 번째 정도의 피보나치 수열의 값을 출력 – 실행 속도 나 메모리 문제가 발생하는지 확인