Study Title _____ Date _____

Chapter5. 함수

5-1 함수 만들기

-함수를 사용하는 것을 함수를 호출한다고 표현하고, 함수를 호출할 때는 괄호 내부에 여러가지 자료를 넣게 되는데, 이러한 자료를 매개변수라 부릅니다.
마지막으로 함수를 호출해서 최종적으로 나오는 결과를 리턴값이라고 합니다.

함수의 기본

-함수의 기본 형태는 다음과 같습니다.

```
def 함수 이름() :
    문장
```

ex)

```
def print_3_times() :
    print("안녕하세요")

print_3_times()
>>안녕하세요
```

함수의 매개변수 만들기

ex)

```
def print_n_times(value, n) :
    for i in range(n) :
        print(value)

print_n_times("안녕하세요", 5)
>>안녕하세요
  안녕하세요
  안녕하세요
  안녕하세요
  안녕하세요
#함수를 호출할때 매개변수를 적게넣거나 더 많이 넣으면 TypeError가 발생합니다
```

가변 매개변수

-print()함수 와 같이 매개변수를 원하는 만큼 받을 수 있는 함수를 가변 매개변수라고 부릅니다.

기본구조는 다음과 같습니다.

```
def 함수이름(매개변수,매개변수,……,*가변 매개변수):
    문장
```

1.가변 매개변수 뒤에는 일반 매개변수가 올 수 없음
2.가변 매개변수는 하나만 사용할수 있음

ex)

```
def pirnt_n_times(n, *values):
    for i in range(n):
        for value in values:
            print(value)

print_n_times(2, "안녕하세요", "즐거운", "파이썬")
>>안녕하세요
  즐거운
  파이썬
  안녕하세요
  즐거운
  파이썬
#여기서 가변매개변수를 리스트처럼 사용했습니다
```

기본 매개변수

-가변 매개변수 뒤에는 일반 매개변수는 올 수 없지만, 기본 매개변수는 올수있으며 형태는 '매개변수=값'의 형태를 띄고 있습니다.이는 매개변수를 입력하지 않았을 경우 매개변수에 들어가는 기본값입니다.

-기본 매개변수 뒤에는 일반 매개변수가 올 수 없습니다.

ex)

```
def print_n_times(value, n=2)
    for i in range(n):
        print(value)

print_n_times("안녕하세요")
>>안녕하세요
  안녕하세요
#기본 매개변수n에 값을 입력하지 않았기 때문에 기본값2가 들어가게됩니다.
```

키워드 매개변수

-가변매개변수와 기본매개변수를 같이 써도 되는지 알아봅시다

```
#기본 매개변수가 가변 매개변수보다 앞에 올때
def print_n_times(n=2, *values):
    for i in range(n):
        for value in values:
            print(value)
print_n_times("안녕하세요", "즐거운", "파이썬")
>>TypeError
```

n에는 "안녕하세요"가 입력되고 values에는 "즐거운","파이썬"이 입력됩니다.

Study Title _____ Date _____

#가변매개변수가 기본매개변수보다 앞에 올때
```
def print_n_times (*values, n=2):
    for i in range(n):
        for value in values:
            print(value)
print_n_times("안녕하세요", "즐거운", "파이썬",3)
>>안녕하세요
  즐거운
  파이썬
  3
  안녕하세요
  즐거운
  파이썬
  3
```
이경우 가변 매개변수가 우선시 됩니다.

#키워드 매개변수
```
def print_n_times(*values, n=2):
    for i in range(n):
        for value in values:
            print(value)
print_n_times("안녕하세요", "즐거운", "파이썬", n=3)
>>
안녕하세요
즐거운
파이썬
안녕하세요
즐거운
파이썬
안녕하세요
즐거운
파이썬
```
-이처럼 매개변수 이름을 지정해서 입력하는것을 키워드 매개변수라고 부릅니다.

리턴

#자료없이 리턴하기

함수 내부에서 return이라는 키워드를 사용할 수있고, 함수가 끝나는 위치를 의미합니다.

ex)
```
def return_test():
    print("A 위치입니다")
    return
    print("B 위치입니다")
return_test()
>>A 위치입니다
```

#자료와 함께 리턴하기

리턴뒤에 자료를 입력하면 자료를 가지고 리턴합니다.

ex)
```
def return_test():
    return 100
value = return_test
print(value)
>>100
```

#아무것도 리턴하지 않기
```
def return_test():
    return
value = return_test
print(value)
>>None
```

기본적인 함수의 활용

-여기서는 리턴을 활용하는것을 살펴보겠습니다.

ex)
```
def sum_all(start, end):
    output = 0
    for i in range(start, end + 1):
        output += i
    return output
print(sum_all(0, 100))
print(sum_all(50, 100))
>>5050
>>3825
```

#기본 매개변수와 키워드 매개변수도 활용해봅시다

ex)
```
def sum_all(start=0, end=100, step=1):
    output=0
    for i in range(start, end + 1, step):
        output += i
    return output
print(sum_all(0, 100, 10))
print(sum_all(end=100, step=2))
>>550
>>2550
```

Study Title _____ Date _____

5-2 함수의 활용

-이번 절에서는 함수가 어떤 식으로 활용되는지 알아보도록 합시다

재귀 함수

-두가지방법으로 팩토리얼 연산자를 구해봅시다

#반복문으로 팩토리얼구하기

ex)

```
def factorial(n):
    output = 1
    for i in range(1, n+1):
        output *= i
    return output

print(factorial(1))
print(factorial(2))
print(factorial(3))
>>1
>>2
>>6
```

#재귀 함수로 팩토리얼 구하기

-재귀란 자기 자신을 호출하는 것을 의미합니다

ex)

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

print(factorial(1))
print(factorial(2))
print(factorial(3))
>>1
>>2
>>6
```

재귀 함수의 문제

-재귀함수는 상황에 따라서 같은 것을 기하급수적으로 많이 반복한다는 문제가 있습니다. 발생하는 문제를 알아보고, 이를 해결할 수 있는 메모화를 알아보겠습니다

#피보나치 수열

-이 수열의 규칙은 다음과 같습니다.
1번째 수열 = 1
2번째 수열 = 1
n번째 수열 = (n-1)번째 수열 + (n-2)번째 수열

ex)

```
def fibonacci(n):
    if n==1:
        return 1
    if n==2:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
print(fibonacci(1))
print(fibonacci(2))
print(fibonacci(3))
print(fibonacci(4))
print(fibonacci(5))
>>1
>>1
>>2
>>3
>>5
```

여기에 fibonacci(50)을 구하면 1시간정도가 걸리게 됩니다. 왜 이렇게 오래 걸리는 걸까요? 코드를 변경해서 문제를 확인해 보겠습니다.

ex)

```
counter = 0
def fibonacci(n):
    print("fibonacci({})를 구합니다.".format(n))
    global counter
    counter += 1

    if n == 1:
        return 1
    if n == 2:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
    fibonacci(10)
    print(counter)
>>fibonacci(10)를 구합니다.
>>fibonacci(9)를 구합니다.
—-생략—
>>fibonacci(2)를 구합니다.
>>fibonacci(1)를 구합니다.
>>109
```

이렇게 덧셈 횟수가 늘어나는 이유는 트리구조이기 때문입니다. 재귀 함수는 한번 구했던 값이라도 처음부터 다시 계산하기 때문에, 계산 횟수가 늘어나는 것입니다.

#unboundLocalError에 대한 처리
-앞의 코드를 보면 global counter라고 되어 있는 부분이 있습니다. 파이썬은 함수내부에서 함수외부에 있는 변수를 참조하지 못합니다. 여기서 참조라는 말은 변수에 접근한다는 의미입니다. 함수내부에서 함수외부에 있는 변수를 참조하려면 다음과 같은 구문을 사용합니다.

```
global 변수 이름
```

global키워드는 파이썬에만 있는 특이한 구조입니다.

#메모화
-재귀함수의 문제는 같은 값을 구하는 연산을 반복하는 것입니다. 따라서 같은 값을 한번만 계산하도록 코드를 수정하면 됩니다.
ex)

```
dictionary = {
    1: 1,
    2: 2
}
def fibonacci(n) :
    if n in dictionary:
        return dictionary[n]
    else:
        output = fibonacci(n-1) + fibonacci(n-2)
        dictionary[n] = output
        return output

print(fibonacci(10))
print(fibonacci(20))
>>89
>>10946
```

이처럼 딕셔너리를 사용해서 한번 계산한 값을 저장합니다. 이를 메모 한다고 표현합니다.

5-3 함수 고급

-이 절에서는 튜플,람다,파일 처리를 살펴볼것입니다. 이는 파이썬만이 가지고 있는 특별한 문법입니다.

튜플

-튜플은 리스트와 비슷한 자료형입니다. 리스트와 다른점은 한번 결정된 요소는 바꿀수 없다는 것입니다. 일반적으로 튜플은 함수와 함께 많이 사용되는 자료형이고 다음과 같이 생성합니다.

```
"""
(data, data, data, …)
“”
```

ex)

```
tuple_test  = (10, 20, 30)
tuple_test[0]
>>10
```

#요소를 변경하려하면 에러가 발생합니다.

```
tuple_test[0] = 1
TypeError : 'tuple' object does not support item assignment
```

#요소 하나만 가지는 튜플
(273) - X
(273,) - O
반드시 쉼표를 넣어줘야함

#괄호 없는 튜플
튜플은 괄호를 생략해도 됩니다.
ex)

```
tuple_test = 10, 20, 30, 40
print(tuple_test)
>>(10, 20, 30, 40)

a, b, c = 10, 20, 30
print(a)
print(b)
print(c)
>>10
>>20
>>30
```

#변수의 값을 교환하는 튜플

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

#튜플과 함수
튜플은 함수의 리턴에 많이 사용합니다. 함수의 리턴에 튜플을 사용하면 여러 개의 값을 리턴하고 할당할 수있기 때문이빈다.
ex)

```
def test() :
    return (10, 20)
a, b = test()
print(a)
print(b)
>>10
>>20
```

Study Title _____ Date _____

람다

-함수라는 '기능'을 매개변수로 전달하는 코드를 더 효율적으로 작성할 수 있도록 파이썬은 람다라는 기능을 제공합니다.

#함수의 매개변수로 함수 전달하기

ex)

```
def call_10_times(func):
    for i in range(10):
        func()
def pirnt_hello():
    print("안녕하세요")

call_10_times(print_hello)
>>
안녕하세요
안녕하세요
안녕하세요
안녕하세요
안녕하세요
안녕하세요
안녕하세요
안녕하세요
안녕하세요
안녕하세요
```

#filter()함수와 map()함수

함수를 매개변수로 전달하는 대표적인 표준함수로 map()함수와 filter()함수가 있습니다.

map()함수는 리스트의 요소를 함수에 넣고 리턴된 값으로 새로운 리스트를 구성해줍니다.
filter()함수는 리스트의 요소를 함수에 넣고 리턴된 값이 True인 것으로, 새로운 리스트를 구성해 주는 함수입니다.

ex)

```
def power(item):
    return item*item
def under_3(item):
    return item < 3
List_input_a = [1, 2, 3, 4, 5]

output_a = matp(power, List_input_a)
print(output_a)
print(list(ouput_a))
>> <map object at 0x03862270>
>> [1, 4, 9, 16, 25]

output_b = filter(under_3, list_input_a)
print(output_b)
print(list(output_b))
>> <filter object at 0x03862290>
>> [1, 2]
```

여기서 결과로 <map object>와 <filter object>가 나오는데 이를 제너레이터라고 부릅니다.

#람다의 개념

매개변수로 함수를 전달하기 위해 함수 구문을 작성하는 것도 번거롭고, 코드 공간 낭비라고 생각했기때문에 람다라는 개념을 생각했습니다.
람다는 간단한 함수를 쉽게 선언하는 방법입니다.

```
""
lambda 매개변수: 리턴값
""
```

ex)

```
power = lambda x: x * x
under_3 = lambda x: x < 3
list_input_a = [1, 2, 3, 4, 5]

output_a = map(power, list_input_a)
print(output_a)
print(list(output_a))

output_b = filter(under_3, list_input_a)
pirnt(output_b)
print(list(output_b))
```

결과는 이전과 같습니다.
코드를 더 단순화하려면 power 나 under_3를 선언할 필요 없이 바로 람다를 표준함수 안으로 넣을 수도 있습니다.

지금까지는 매개변수가 하나인 람다만 살펴 보았는데, 다음과 같이 매개변수가 여러 개인 람다도 만들 수 있습니다.

```
"""
lambda x, y: x * y
""
```

Study Title _____ Date _____

파일처리

-파일을 처리하려면 일단 파일열기를 해야합니다. 파일을 열면 파일 읽기 또는 파일 쓰기를 할 수 있습니다.

#파일 열고 닫기

-파일을 열 때는 open()함수를 사용합니다.

“””
파일객체 = open(문자열:파일경로, 문자열:읽기모드)
“””

모드에는 다음과 같은 것들이있습니다.
w-새로쓰기모드
a-뒤에 이어서 쓰기 모드
r-읽기 모드

파일을 닫을 때는 close()함수를 하용합니다.
파일객체.close()

ex)

```
file = open(“basic.txt”, “w”)
file.write(“Hello Python Programming..!”)
file.close()
```

프로그램을 실행하면 프로그램과 같은 폴더에 basic.txt라는 파일이 생성되고 파일을 열어보면 메모장으로 다음과 같은 문구과 입력이 되어있습니다.

#with 키워드

-파일은 열면 반드시 닫아주어야합니다. 하지만 코드가 길어지면 닫지않는 실수를 할 수도 있기때문에 이를 방지하기위해 with키워드라는 기능이 있습니다.

“””
with open(문자열:파일경로,문자열:모드)as파일객체:
    문장
“””

위에 있던 코드를 with을 사용해 수정하면 다음과 같습니다.

ex)

```
with open(“basic.txt”, “w”) as file:
    file.write(“Hello Python Programming..!”)
```

이러면 결과는 위와 같습니다.

#스트림

-프로그램이 외부파일,외부네트워크 등과 통신할 때는 데이터가 흐르는 길을 만들어야합니다. 이를 스트림이라 부릅니다. open()함수는 프로그램에서 파일로 흐르는 길을 만드는 것이고 close()함수는 길을 닫는 것입니다.

#텍스트 읽기

-파일에 텍스트를 쓸 때는 방금 살펴보았던 것처럼 write()함수를 사용합니다. 반대로 파일을 읽을 때는 read()함수를 사용합니다.

“””
파일객체.read()
“””

파일을 열고 파일 객체의 read()함수를 호출하기만 하면 내부에 있는 데이터를 모두 읽어 출력합니다.

ex)

```
with open(“basic.txt”, “r”) as file:
    contents = file.read()
print(contents)
>>Hello Python Programming..!
```

텍스트 한 줄씩 읽기

-텍스트를 사용해 데이터를 구조적으로 표현할 수 있는 방법으로 CSV,XML,JSON 등이 있습니다. 이중에서 CSV를 간단하게 살펴봅시다. CSV는 comma separated Values의 줄임말로 쉼표로 구분된 값을 의미합니다.

ex)

```
이름, 키, 몸무게
윤인성, 176, 62
연하진, 169, 50
```

CSV는 한 줄에 하나의 데이터를 나타내며, 각각의 줄은 쉼표를 사용해 데이터를 구분합니다.
일단 간단한 코드로 1000명의 이름,키,몸무게 데이터를 만들어 봅시다.

ex)

```
import random
hanguls = list(“가나다라마바사아자차카타파하”)
with open(“info.txt”, “w”) as file:
    for i in range(1000):
        name = random.choice(hanguls)\
            + random.choice(hanguls)
        weight = random.randrange(40, 100)
        height = random.randrange(140, 200)

        file.write(“{}, {}, {}\n”.format(name,\
        weight, height))
```

이제 데이터가 많이 있기때문에 한 줄씩 읽어봅시다. 데이터를 한 줄씩 읽어 들일 때는 for반복문을 다음과 같은 형태로 사용합니다.

“””
for 한 줄을 나타내는 문자열 in 파일 객체:
    처리
“””

Study Title _____ Date _____

위에서 만든 데이터를 한 줄씩 읽으면서 BMI를 계산해 봅시다.
ex)
```
with open("info.txt", "r") as file:
    for line in file:
        (name,weight,height) = line.strip().split(",")
    if (not name) or (not weight) or ( not height):
        continue
    bmi = int(weight)/((int(height)/100)**2)
    result = ""
    if 25 <= bmi:
        result = "과체중"
    elif 18.5 <= bmi:
        result = "정상 체중"
    else:
        result = "저체중"

    pritn('\n'.join([
        "이름: {}",
        "몸무게: {}",
        "키: {}",
        "BMI: {}",
        "결과: {}"
    ]).format(name, weight, height, bmi, result))
```

```
>>
이름 : 나자
몸무게 : 82
키 : 193
BMI : 22.0810428019
결과 : 정상체중
—반복—
```

#제너레이터
-제너레이터는 파이썬의 특수한 문법 구조입니다. 제너레이터는 이터레이터를 직접 만들 때 사용하는 코드입니다. 함수 내부에 yield키워드를 사용하면 해당 함수는 제너레이터 함수가 되며, 일반함수와는 달리 함수를 호출해도 함수 내부의 코드가 실행되지 않습니다.
ex)
```
def text():
    print("함수가 호출되었습니다.")
    yield "test"
print("A 지점 통과")
test()
print("B 지점 통과")
print(test())
```

```
>>
A 지점 통과
B 지점 통과
<generator object test at 0x02F20C90>
```

제너레이터 함수는 제너레이터를 리턴하기때문에, 문자열이 출력되지 않습니다.

제너레이터 객체는 next()함수를 사용해 함수 내부의 코드를 실행합니다. 이때 yield키워드 부분까지만 실행하며, next()함수의 리턴값으로 yield키워드 뒤에 입력한 값이 출력됩니다.
ex)
```
def test():
    print("A지점 통과")
    yield 1
    print("B지점 통과")
    yield2
    print("C지점 통과")

output = test()
print("D지점 통과")
a = next(output)
print(a)
print("E지점 통과")
b = next(output)
print(b)
print("F지점 통과")
c = next(output)
print(c)
next(output)
```

```
>>
D지점 통과
A지점 통과
1
E지점 통과
B지점 통과
2
F지점 통과
C지점 통과
StopIteration
```

next()함수를 호출한 이후 yield키워드를 만나지 못하고 함수가 끝나면 StopIteration이라는 예외가 발생합니다. 이처럼 제너레이터 객체는 함수의 코드를 조금씩 실행할때 사용합니다. 메모리의 효율성을 위해서 입니다.