A photograph of a baboon sitting at a desk, looking directly at the camera. The baboon has greyish-brown fur and a long, light-colored beard. It is positioned in front of a computer monitor and keyboard. The background is a reddish-brown wall.

원숭이도 이해하는
Python Pandas

A photograph of a monkey sitting at a desk, using a laptop. The monkey is positioned on the left side of the frame, facing right. Its right hand is on the laptop keyboard, and its left hand is resting on the desk. The laptop is open, and the keyboard is visible. The background is a solid red color. The word "Python" is overlaid in the center of the image in a large, white, sans-serif font.

Python

변수

- 파이썬의 변수는 항상 선언과 동시에 값을 초기화를 해야한다.
- 변수의 데이터 타입은 초기화되는 값의 타입이다.
- ‘,’ 를 사용해 동시에 여러 개의 변수를 초기화 시킬 수 있다.

```
1  a = 2 # int
2  b = 2.0 # float
3  c = "hello" # string
4  v1, v2, v3 = 1, 2, 3
5  # v1 == 1, v2 == 2 , v3 == 3
6
```

출력

- 파이썬에서 출력은 print 함수를 사용한다.
- default → `print(*value, sep=" ", end="\n")`
 - ✓ *value는 가변 인자: 몇개를 넣든 상관없이 튜플로 묶어서 전달
- value에 들어온 값들을 출력한다.
- “,”를 사용하여 여러 개의 값을 value에 전달할 수 있다.
- “,”를 기준으로 두 개의 값 사이에 sep의 값을 넣어준다.
- 모든 출력이 끝나면 end의 값을 출력한다.
- sep, end 요소는 따로 설정할 수 있다.

```
1 print(2)
2 print(2,3,4)
3 print(2,3,4,sep="*")
4 print(2,3,4,sep=" | ",end="hello")
```

```
2
2 3 4
2*3*4
2 | 3 | 4hello
```


입력

- 파이썬에서 입력은 input() 함수를 사용한다.
- 입력 값을 저장할 변수 = input(string=" ")
- string에 값(어떤 타입이든 가능)을 전달하면 키보드 입력 전에 입력 받은 값을 출력하고 키보드 입력 대기를 한다.
- 반환되는 값은 항상 str(문자열)이다.

```
a = input(1)
print(type(a))
```

```
12
<class 'str'>
```

```
a = input(str(2)+"hello")
print(type(a))
```

```
2hello2.0
<class 'str'>
```

```
a = input([1, 2, 3, 4])
print(type(a))
```

```
[1, 2, 3, 4]hello
<class 'str'>
```

데이터 타입

- 정수형 : int → 1 , 2 , 3 , 4 , 0b1101(2진수), 0o52(8진수), 0x2a(16진수)
- 실수형 : float → 1.2 , 2.000 , 1.12314
- 문자열 : str(파이썬은 문자 타입이 없음!!) → “1” , ‘hello’ , ““ hello ”” , ““ bye ””
- 논리형 : bool → True , False
- 자료구조형 : list(리스트), tuple(튜플), dictionary(사전), set(집합)

```
print(type(2))
print(type(2.0))
print(type("h"))
print(type(True))
print(type([1, 2, 3]))
print(type((1, 2, 3)))
print(type({"1": 1, "2": 2, "3": 3}))
print(type({1, 2, 3}))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
<class 'list'>
<class 'tuple'>
<class 'dict'>
<class 'set'>
```

타입 변환 함수

- 변수의 타입을 함수를 통해 변경할 수 있다.
- 형식에 맞지 않는 경우는 변환이 안될 수 있다.
 - Ex) 1을 사전으로 → X 키:값 쌍이 아니라 불가능
- 정수 : 변환된 값=int(변환 할 변수)
- 실수 : 변환된 값=float(변환 할 변수)
- 문자열 : 변환된 값=str(변환 할 변수)
- 논리값 : 변환된 값=bool(변환 할 변수)
- 리스트 : 변환된 값=list(변환 할 변수)
- 튜플 : 변환된 값=tuple(변환 할 변수)
- 사전 : 변환된 값=dict(변환 할 변수)
- 집합 : 변환된 값=set(변환 할 변수)

```
1 print(type(int("2")))  
2 print(type(float("2.2")))  
3 print(type(str(2)))  
4 print(type(bool("True")))  
5 print(type(list("[1,2,3,4]")))  
6 print(type(tuple([1, 2, 3, 4])))  
7 print(type(dict([(1, 2), (2, 3)])))  
8 print(type(set([1, 2, 3, 4, 5])))
```

```
<class 'int'>  
<class 'float'>  
<class 'str'>  
<class 'bool'>  
<class 'list'>  
<class 'tuple'>  
<class 'dict'>  
<class 'set'>
```

문자열

- 파이썬의 문자열은 메모리상에 문자들이 일렬로 저장되어 있는 형태이다.
- 초기화 되면 인덱싱으로 변경 불가능 하다.(접근하는 것은 가능)
- list 처럼 slicing, indexing이 가능하다.
- + , * 연산이 가능하지만 + 연산은 문자열 끼리만 가능하다.

```
a = "hello"
print(a[0])
a[0] = "hi"
```

```
a = "hello"
print(a+3)
```

```
Traceback (most recent call last):
  File "/Users/pupba/Desktop/workspace/test/python/test.py", line 1, in <module>
    print(a+3)
TypeError: can only concatenate str (not "int") to str
```

```
h
Traceback (most recent call last):
  File "/Users/pupba/Desktop/workspace/test/python/test.py", line 1, in <module>
    a[0] = "hi"
TypeError: 'str' object does not support item assignment
```

```
a = """
hello
안녕
"""
b = "하세요"
print(a+b)
print(b*3)
```

```
hello
안녕
하세요
하세요하세요하세요
```


문자열 관리

- `upper()` : 문자열을 모두 대문자로 만들어주는 메서드이다.
- `lower()` : 문자열을 모두 소문자로 만들어주는 메서드이다.
- `len()` : 문자열의 길이를 반환하는 내장 함수 이다.
- `swapcase()` : 대문자는 소문자로 소문자는 대문자로 바꿔주는 메서드 이다.
- `title()` : 문장의 제일 앞에만 대문자로 바꿔주는 메서드이다.

```
st = 'Python hello'
print(st.upper())
print(st.lower())
print(st.swapcase())
print(st.title())
```

```
PYTHON HELLO
python hello
pYTHON HELLO
Python Hello
```

문자열 관리

- `split()` : 문자열을 매개변수(없을 시 공백) 기준으로 잘라서 리스트로 반환하는 메서드이다.
- `replace(원본 값, 바꿀 값)` : 앞에 매개변수를 뒤에 매개변수로 바꿔주는 메서드이다.
- `join()` : 매개변수로 받은 리스트(요소의 타입은 str) 요소 사이에 구분자(.앞에 있는 값)를 넣어 그 리스트를 반환하는 메서드

```
st1 = "안 녕 하 세 요"
l1 = st1.split()
print(l1)
rs = st1.replace('안', '한')
print(rs)
['안', '녕', '하', '세', '요']
한 녕 하 세 요
```

```
l1 = "1234"
j1 = "|".join(l1)
print(j1)
1|2|3|4
```

문자열 관리

- `strip()`, `rstrip()`, `lstrip()` : 구분자(.앞에 값)인 문자열에서 공백을 제거하는 메서드이다. `strip`은 양쪽, `rstrip`은 오른쪽만, `lstrip`은 왼쪽만 제거한다. 매개변수로 문자열을 주면 해당 문자열을 양쪽(오른쪽, 왼쪽)에서 찾아서 제거한다.

```
str1 = "    하이    "  
str2 = "00000하이00000"  
print(str1.strip())  
print(str1.rstrip())  
print(str1.lstrip())  
print(str2.strip("0"))  
print(str2.rstrip("0"))  
print(str2.lstrip("0"))
```

```
하이  
|      | 하이  
하이  
하이  
00000하이  
하이00000
```

문자열 관리

- **center()** : 문자열 가운데 정렬, 숫자를 매개변수로 전달 시 양쪽에 문자열 개수를 제외한 길이를 반반 나눠서 공백 만든다. 두번째 매개변수를 주면 그 값으로 공백을 대체한다.
- **ljust()** : 문자열을 왼쪽으로 정렬한다. 숫자를 매개변수로 전달 시 오른쪽에 문자열 개수를 제외한 길이 만큼 공백을 만든다.
- **rjust()** : 문자열을 오른쪽으로 정렬한다. 숫자를 매개변수로 전달 시 왼쪽에 문자열 개수를 제외한 길이 만큼 공백을 만든다.
- **zfill()** : 문자열을 오른쪽으로 정렬한다. 숫자를 매개변수로 전달 시 오른쪽에 문자열 개수를 제외한 길이 만큼 0을 채워 넣는다.

문자열 관리

- `isdigit()` : 문자열이 숫자로 이뤄져 있는지 판별하는 메서드이다.
- `isalpha()`: 문자열이 알파벳으로 이뤄져 있는지 판별하는 메서드이다
- `isalnum()` : 문자열이 숫자 또는 알파벳으로 이뤄져 있는지 판별하는 메서드이다
- `islower()` : 문자열이 소문자로 이뤄져 있는지 판별하는 메서드이다.
- `isupper()` : 문자열이 대문자로 이뤄져 있는지 판별하는 메서드이다.
- `isspace()` : 문자열이 공백으로 이뤄져 있는지 판별하는 메서드이다.

```
st = 'hello'
print(st.center(10))
print(st.center(10, '-'))
print(st.rjust(10))
print(st.ljust(10))
print(st.zfill(10))
```

| |
|-----------|
| hello |
| --hello-- |
| hello |
| hello |
| 0000hello |

```
st = '123ab'
print(st.isdigit())
print(st.isupper())
print(st.islower())
print(st.isalnum())
print(st.isspace())
print(st.isalpha())
```

| |
|-------|
| False |
| False |
| True |
| True |
| False |
| False |

리스트

- []안에 여러가지 데이터 타입의 값들이 저장된 형태의 자료구조이다.
- 빈 리스트를 만드는 방법 → `a = []` or `a = list()`
- 다른 데이터 타입이 들어가도 상관 없다.
 - Ex) `a = [1, "2", 2.0, True, [1,2,3], (1,2), {1,2,3}, {"hello":2}]`
- C, C++, Java의 n배열 처럼 n차원 리스트 형태를 제공한다.
 - 2차원 리스트 → `a = [[1,2,3],[2,3,4],[4,5,6]]`

```
a = []  
b = list()  
print(a, b)
```

```
a = [1, "2", 2.0, True, [1, 2, 3],  
(1, 2), {1, 2, 3}, {"hello": 2}]  
print(type(a))  
print(a)
```

```
<class 'list'>  
[1, '2', 2.0, True, [1, 2, 3], (1, 2), {1, 2, 3}, {'hello': 2}]
```

```
a = [[1, 2, 3], [2, 3, 4]]  
print(a)  
print(a[0])  
print(a[0][2])
```

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

리스트 관리

- 리스트 명[인덱스] = 수정할 값 : 리스트 수정하는 방법이다.
- del 리스트 명[인덱스] : 리스트 삭제, slicing으로 한꺼번에 삭제 가능하다.
- append(추가할 값) : 리스트 제일 뒤에 요소를 추가하는 메서드 이다.
- insert(인덱스,추가할 값) : 리스트에 매개변수로 준 인덱스의 요소 앞에 값을 추가한다.
- remove(삭제할 값) : 값에 해당하는 리스트의 요소를 삭제하는 메서드

```
a = [1, 2, 3, 4, 5]
a[0] = 5
print(a)
del a[2:]
print(a)
a.append(7)
print(a)
```

```
[5, 2, 3, 4, 5]
[5, 2]
[5, 2, 7]
```

```
a = [1, 2, 3, 4, 5]
a.insert(2, [1, 2])
print(a)
a.remove([1, 2])
print(a)
```

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

slicing

- slicing 기법으로 리스트나 튜플의 부분을 추출 할 수 있다.
- [begin:end:step]의 구조를 가지고 있고 begin부터 end-1까지 step만큼 인덱스를 증가시키면서 리스트 요소들을 추출한다.
- begin 생략 시 0부터 시작, end 생략 시 끝 까지, step 생략 시 1씩

```
a = [1, 2, 3, 4, 5, 6, 7]
print(a[3:5:2])
print(a[:3])
print(a[3:])
```

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

```
b = (1, 2, 3, 4, 5)
print(b[0:3])
print(b[::2])
print(b[3:])
```

```
(1, 2, 3)
(1, 3, 5)
(4, 5)
```

리스트 관리

- `pop()` : 마지막 요소를 제거하고 리턴 하는 메서드 이다. 매개변수로 인덱스를 넘기면 해당하는 인덱스의 값을 제거하고 리턴 한다.
- `index(값)` : 값에 해당하는 인덱스를 리턴 하는 메서드 이다. 값이 2개 이상이라면 오른쪽 기준으로 제일 먼저 만나는 값의 인덱스를 리턴 한다.
- `sort()` : 리스트의 요소들을 오름차순으로 정렬해주는 메서드 이다. `reverse` 속성을 `True`로 주면 내림차순으로 정렬된다.
- `reverse()` : 리스트의 요소들을 거꾸로 바꿔주는 메서드이다.
- `count(값)` : 리스트에 포함 된 값들의 개수를 세주는 메서드이다.

리스트 관리

- 값 in 리터럴 변수(list,tuple 등) : 값이 리터럴 변수 안에 있으면 참을 반환하는 연산자
- 값 not in 리터럴 변수 : 값이 리터럴 변수에 없으면 참을 반환하는 연산자
- sorted() : 리스트의 값을 오름차순으로 정렬해서 반환하는 내장 함수, reverse 속성이 True이면 내림차순 정렬, key 속성에 값을 주면 그 값에 해당하는 기준으로 정렬한다.

```
a = [1, 2, 3, 4, 5]
print(a.pop(), a.pop(2))
print(a.index(1))
a.sort(reverse=True)
print(a)
print(a.count(2))
```

```
5 3
0
[4, 2, 1]
1
```

```
a = [1, 2, 3, 4, 5]
if 2 in a:
    print(2, "가 있습니다.")
if 6 not in a:
    print(6, "가 없습니다.")
sortA = sorted(a, reverse=True)
print(sortA, a)
```

```
2 가 있습니다.
6 가 없습니다.
[5, 4, 3, 2, 1] [1, 2, 3, 4, 5]
```


리스트의 복사

- `a = [1,2,3]` 이라는 리스트가 있을 때 `a=b`를 하게 되면 변수 `a`, `b`는 같은 리스트를 공유하게 된다.
- 이렇게 되면 `b`를 수정하고 `a`를 출력하면 수정한 결과가 나오기 때문에 메모리를 복사하는 깊은 복사가 필요하다.
- 깊은 복사 방법
 1. `copy()` 메서드 : 복사할 변수 = 리스트.`copy()`
 2. `[:]` 리스트 슬라이싱 : 복사할 변수 = 리스트`[:]`

```
l1 = [1, 2, 3, 4]
l2 = l1
l2[2] = 2
print(l1)
```

```
[1, 2, 2, 4]
```

```
l1 = [1, 2, 3, 4]
cl1 = l1.copy()
cl2 = l1[:]
cl1[2] = 2
cl2[3] = 1
print(l1)
print(cl1)
print(cl2)
```

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

튜플

- ()안에 여러가지 데이터 타입의 값들이 저장된 형태의 자료구조이다.
- 빈 튜플을 만드는 방법 → `a = ()` or `a = tuple()`
- 튜플은 인덱싱 등으로 값에 접근은 가능하지만 값 수정은 불가능하다.
- 다른 데이터 타입이 들어가도 상관 없다.
- C, C++, Java의 n배열 처럼 n차원 튜플 형태를 제공한다.

```
a = (1, "2", [1, 2], (1, 2))  
b = (1,)  
print(a[1])  
print(a, b)
```

```
2  
(1, '2', [1, 2], (1, 2)) (1,)
```

```
a = (1, 2, 3)  
a[0] = 2
```

```
File ~/Users/papad/Desktop/workspace/test/python/test.py  
a[0] = 2  
TypeError: 'tuple' object does not support item assignment
```

```
a = ((1, 2, 3), (4, 5, 6))  
print(a[0][0:1])
```

```
(1,)
```

사전

- 파이썬의 사전은 {key:value}의 형태로 여러 데이터 타입이 저장되는 자료구조이다.
- key값과 value 값은 항상 쌍으로 있어야하며 둘 중 하나만 있을 수는 없다.
- key 값은 int, float, str, bool, tuple 타입의 값만 가능하다.(고유한 값만 가능!)
- value 값은 어떤 타입이든 모두 가능하다.
- {} 안에 key : value 쌍의 순서가 상관 없으므로 일반적인 인덱싱이 불가능하다.
- value 을 호출 하려면 그 쌍이 되는 key의 값을 “사전 명[key값]”를 호출한다.
- value 값을 수정할 때는 “사전 명[key값] = 수정할 값” 형태로 사용한다.
- 같은 key 값이 있을 경우 제일 뒤에 key값의 value로 설정한다.

```
a = {"1": 2, 2: [1, 2, 3], 3.5: (1, 2, 3), True: "1"}
print(a["1"], a[2], a[3.5], a[True])
a["1"] = [1,2,3]
print(a["1"])
```

```
[Running] python3 -u 70
2 [1, 2, 3] (1, 2, 3) 1
[1, 2, 3]
```

사전에 사용되는 메서드

- `dict()` : 사전으로 만들어준다.
- `keys()` : 사전의 키 값들을 리스트 형태로 반환한다.(리스트는 아님)
- `values()` : 사전의 value 값들을 리스트 형태로 반환한다.(리스트는 아님)
- `items()` : 사전의 key, value 쌍을 튜플로 묶은 값들을 리스트 형태로 반환(리스트는 아님)

```
a = {"1": 2, 2: [1, 2, 3], 3.5: (1, 2, 3), (1, 2, 3): "1"}
print(a.keys())
print(a.values())
print(a.items())
```

```
a = [("Tom", 2), ("Bob", 4), ("Alice", 6)]
dic = dict(a)
print(dic)
```

```
{'Tom': 2, 'Bob': 4, 'Alice': 6}
```

```
dict_keys(['1', 2, 3.5, (1, 2, 3)])
dict_values([2, [1, 2, 3], (1, 2, 3), '1'])
dict_items([('1', 2), (2, [1, 2, 3]), (3.5, (1, 2, 3)), ((1, 2, 3), '1')])
```

집합

- 사전과 같은 {}로 묶지만 값만 저장된다.
- 중복을 허락하지 않는다.
- set()로 다른 데이터 타입을 사전으로 변환 시킬 수 있다.
- 수학의 집합과 같이 합집합, 차집합, 부분집합 등의 연산 또는 메서드들을 지원한다.
- 순서가 의미가 없어 인덱싱이 불가능하다.

```
a = {1, 2, 3, 4}
b = [1, 2, 3, 4]
c = (1, 2, 3, 4)
print(a, set(b), set(c))
```

```
{1, 2, 3, 4} {1, 2, 3, 4} {1, 2, 3, 4}
```

```
a = [1, 1, 3, 3, 2, 2, 2, 4]
print(set(a))
```

```
{1, 2, 3, 4}
```


사전의 연산

- 교집합 : `set1 & set2` or `set1.intersection(set2)`
- 합집합 : `set1 | set2` or `set1.union(set2)`
- 차집합 : `set1 - set2` or `set1.difference(set2)`
- 대칭 차집합 : `set1 ^ set2` or `set1.symmetric_difference(set2)`

```
set1 = {1, 2, 3, 4, 5}
set2 = {2, 3, 4, 5, 6}
print(set1 & set2)
print(set1 | set2)
print(set1 - set2)
print(set1 ^ set2)
```

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

```
set1 = {1, 2, 3, 4, 5}
set2 = {2, 3, 4, 5, 6}
print(set1.intersection(set2))
print(set1.union(set2))
print(set1.difference(set2))
print(set1.symmetric_difference(set2))
```

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

기타 중요 함수

- `zip(coll1, coll2)` : 두 컬렉션 리터럴 변수를 튜플 쌍으로 합쳐서 반환 하는 내장 함수이다.
- `map(fuc,coll1)` : 컬렉션 리터럴 변수의 요소들 하나하나에 `fuc(함수)`를 적용하여 반환하는 내장 함수이다.
- `len()` : 매개변수로 받은 컬렉션 리터럴 변수의 길이를 반환하는 내장 함수 이다.

```
a = [1, 2, 3, 4]
b = [5, 6, 7, 8]
zipList = zip(a, b)
print(list(zipList))
[(1, 5), (2, 6), (3, 7), (4, 8)]
```

```
a = [1, 2, 3, 4]
print(list(map(str, a)))
['1', '2', '3', '4']
```

```
l1 = [1, 2, 3, 4, 5]
print(len(l1))
5
```

조건문

- **if** : 파이썬의 조건문의 실행 명령은 블록 단위로 구분된다.
- **if-else** : if문이 참일 때는 if 아래 블록 명령이 실행되고 거짓일 경우 else 아래 블록 명령이 실행된다.
- **elif** : 다른 언어의 if-else if 문과 같은 동작을 한다. if문과 elif가 같이 쓰이면 if부터 조건을 검사하면서 제일 먼저 만족하는 조건을 실행한다.

```
if 12 > 2:  
    print("if")  
if
```

```
if 1 > 2:  
    print("if")  
else:  
    print("else")  
else
```

```
a = 10  
if a > 2:  
    print("if")  
elif a > 4:  
    print("else")  
if
```

```
a = 10  
if a > 2:  
    print("if")  
if a > 4:  
    print("else")  
if  
else
```

반복문

- **while loop** : 조건이 참일 때 동안 반복하는 반복문 이다. 보통 제어 변수를 선언하여 같이 사용한다.
- **break** : 반복을 멈추는 키워드, 반복문 내부에서 이 키워드를 만나면 그 시점에서 반복문이 종료된다.
- **continue** : 다음 loop로 넘어가는 키워드, 반복문 내부에서 이 키워드를 만나면 그 시점에서 바로 다음 반복으로 넘어간다.
- **pass** : 아무 의미 없는 키워드, 보통 반복문 내부 작성 전에 대기 용으로 적어 넣는다.

```
a = 1
while(a < 5):
    print(a, end=" ")
    a += 1
```

1 2 3 4

```
a = 1
while(True):
    if a == 5:
        break
    elif a == 3:
        a += 1
        continue
    print(a, end=" ")
    a += 1
```

1 2 4

```
a = 1
while(a == 3):
    pass
```

반복문

- for loop : in을 기준으로 오른쪽의 컬렉션 리터럴 변수의 요소들을 하나씩 꺼내서 왼쪽의 변수에 넣는 행위를 반복하는 loop, 반복 횟수는 컬렉션 리터럴 변수의 요소의 개수 만큼 반복한다. 다중 반복문을 지원한다.

```
l1 = [1, 2, 3, 4, 5]
for i in l1:
    print(i, end=" ")
1 2 3 4 5
```

```
s1 = {1, 2, 3, 4, 5}
for i in s1:
    print(i, end=" ")
1 2 3 4 5
```

```
d1 = {"a": 1, "b": 2, "c": 3}
for a in d1:
    print(a, end=" ")
a b c
```

```
t1 = (1, 2, 3, 4, 5)
for i in t1:
    print(i, end=" ")
1 2 3 4 5
```

```
l1 = [1, 2, 3, 4]
for i in l1:
    for j in l1:
        print(i, j, end=" ")
1 1 1 2 1 3 1 4 2 1 2 2 2 3 2 4 3 1 3 2 3 3 3 4 4 1 4 2 4 3 4 4
```


for loop 활용

- `range(start=0,end,step=1)` : start부터 end-1까지 step만큼 증가하는 리스트 형태를 반환하는 내장함수이다.
- `zip`함수로 반복문 제어 변수 2개를 사용할 수 있다.
- 문자열을 하나하나 출력 할 수 있다.

```
for i in range(5):  
    print(i, end=' ')  
for i in range(1, 5):  
    print(i, end=' ')
```

0 1 2 3 4 1 2 3 4

```
for a, b in zip(range(5), range(5)):  
    print(a, b)
```

0 0
1 1
2 2
3 3
4 4

```
for i in "hello":  
    print(i, end=" ")
```

h e l l o

for loop 활용

- `enumerate()` : 컬렉션 리터럴 변수의 요소와 인덱스를 튜플로 묶어서 반환하는 내장 함수 이다.
- `reversed()` : 컬렉션 리터럴 변수의 요소들을 거꾸로 바꿔서 반환하는 내장 함수 이다.
- 리스트 내포 : 반복문의 제어 변수 값들로 리스트를 한번에 생성 할 수 있다.

```
a = [1, 2, 3, 4, 5]
for idx, i in enumerate(a):
    print(idx, i)
```

```
0 1
1 2
2 3
3 4
4 5
```

```
a = [1, 2, 3, 4, 5]
for i in reversed(a):
    print(i, end=' ')
```

```
5 4 3 2 1
```

```
a = [n*2 for n in range(5)]
print(a)
[0, 2, 4, 6, 8]
```

```
a = [n for n in range(5)]
print(a)
[0, 1, 2, 3, 4]
```

```
a = [n for n in range(5) if n > 2]
print(a)
[3, 4]
```

전역변수와 지역변수

- 전역변수 : 프로그램이 끝날 때 까지 사라지지 않고 프로그램 내부 어디서나 호출 가능한 변수, 프로그램 종료와 동시에 사라지는 변수이다.
- 지역변수 : 블록 안에서 선언되는 변수로 그 블록을 벗어나면 사라진다.
- 파이썬에서 블록에서 전역변수를 사용하려면 global 예약어를 사용한다.

```
a = 10
print(a)
def tmp():
    global a
    a = 20

tmp()
print(a)
```

```
a = 10
print(a)
def tmp():
    a = 20

tmp()
print(a)
```

함수

- 파이썬에서의 함수 선언은 def로 시작한다.

- 정의

- ✓ def 함수명(매개변수) :
실행 코드
return 반환할 값

- 호출

- ✓ 변수 = 함수명(매개변수)

- 반환하는 값이 필요 없을 경우 return을 사용하지 않아도 된다.
- 매개변수는 ,를 기준으로 몇개를 써도 상관 없다.
- 매개변수의 타입은 변수와 마찬가지로 전달 받는 값의 타입으로 초기화 된다.
- 디폴트 매개변수를 지원한다.

```
def printHello(str1):  
    print(str1)  
  
def reAdd(a, b):  
    return a+b  
  
printHello("hello")    hello  
print(reAdd(1,2))      3
```

함수 응용

- 가변 인수로 매개변수를 받게 되면 전달되는 모든 값들이 튜플로 묶어서 전달이 된다.
- 키워드 가변 인수로 매개변수를 받게 되면 함수 정의 시 정의한 key값과 그 key값의 value 값의 쌍이 사전으로 만들어져서 전달 된다.
- 람다 함수를 사용하면 간단한 함수를 바로 만들어 변수 처럼 사용할 수 있다.
- lambda 변수(디폴트 매개변수 사용 가능) : 계산식(리턴 값)

```
def sumAll(*args):  
    a = [n for n in args]  
    return sum(a)  
  
print(sumAll(1, 2, 3, 4, 5, 6))
```

21

```
def sumABC(**args):  
    a = args['a']  
    b = args['b']  
    c = args['c']  
    return a+b+c  
  
print(sumABC(a=1, b=3, c=5))
```

9

```
a = lambda x=2, y=2: x+y  
print(a())
```

4

예외 처리

- 파이썬에서 오류 발생 시 그 오류를 처리 해주는 기법이 있다.
- **try** : 일반 실행 블록이다.
- **except** : try 블록 수행 중 오류 발생 시 블록 실행, 오류가 없을 시 실행되지 않는다.
- **finally** : 예외 상관 없이 반드시 수행되는 블록이다.
- **else** : 예외가 없을 경우 수행되는 블록이다.
- 여러 개의 except를 사용하여 여러 개의 예외를 처리 할 수 있다.
- “raise 예외 이름”으로 오류를 발생 시킬 수 있다.

```
[running] python3  
ZeroDivisionError  
finally
```

```
try:  
    raise ZeroDivisionError  
except ZeroDivisionError:  
    print("ZeroDivisionError")  
else:  
    print("else")  
finally:  
    print("finally")
```

예외의 종류

- `KeyError` : 없는 Key 값에 접근하려고 하면 발생
- `ValueError` : 부적절한 값을 가진 인자를 받았을 때, 참조 값이 없을 때 발생
- `SyntaxError` : 문법 오류일 때 발생
- `NameError` : 지역 변수, 전역 변수의 이름을 찾을 수 없는 경우 발생
- `ZeroDivisionError` : 0으로 나누려는 경우 발생
- `FileNotFoundError` : 존재하지 않는 파일이나 디렉토리에 접근할 때 발생
- `TypeError` : 서로 다른 타입으로 연산하려고 할 때 발생
- `AttributeError` : 잘못된 메서드나 속성을 호출하거나 대입했을 때 발생
- `ConnectionError` : 서버를 켜지 않았을 때 발생

제너레이터와 yield

- yield 문 : 함수를 종료하지 않으면서 값을 계속 반환하는 예약어 이다.
- 제너레이터(생성자) 함수 : 리터럴을 생성해준다. yield가 포함된 함수이다.

```
def genFuc():  
    yield 1  
    yield 2  
    yield 3  
print(list(genFuc()))
```

[1, 2, 3]

```
def genFuc(num):  
    for i in range(0, num):  
        yield i  
        print('제너레이터 진행중')  
  
for data in genFuc(5):  
    print(data)
```

0
제너레이터 진행중
1
제너레이터 진행중
2
제너레이터 진행중
3
제너레이터 진행중
4
제너레이터 진행중

함수의 집합 모듈

- “import 모듈명”으로 .py 파일의 함수들을 읽어와 사용할 수 있다.
- “from 모듈명 import 함수 이름”를 사용하면 함수 호출 시 모듈명을 생략할 수 있다.
- 모듈에는 파이썬에서 제공하는 “표준 모듈”, 사용자가 직접 만들어 사용하는 “사용자 정의 모듈”, 외부나 회사 단체에서 제공하는 “서드 파티(3rd Party) 모듈”이 있다.
- 서드 파티 모듈로 사람이 상상하는 거의 모든 기능을 구현 할 수 있다.

mod1.py ×

mod1.py > ...

```
1  def sumAll(list):  
2      tmp = [n for n in list]  
3      return sum(tmp)
```

```
import mod1  
a = [1, 2, 3, 4, 5]  
print(mod1.sumAll(a)) 15
```

```
from mod1 import sumAll  
a = [1, 2, 3, 4, 5]  
print(sumAll(a)) 15
```

표준 모듈

- 파이썬 인터프리터를 설치하면 자동으로 같이 설치되는 모듈들 이다.
- 따로 설치나 경로 지정을 하지 않아도 사용가능하다.
- import sys
print(sys.builtin_module_names) 으로 제공되는 표준 모듈의 목록을 확인 할 수 있다.
- import 모듈명
dir(모듈명) 으로 모듈이 제공하는 함수 목록을 확인 할 수 있다.

```
import sys
print(sys.builtin_module_names)
['_abc', '_ast', '_codecs', '_collections', '_func'
```

```
import math
print(dir(math))
['__doc__', '__file__', '__loader__', '__name__',
```

사용자 모듈

- .py 파일을 직접 작성하여 모듈로 사용한다.
- 모듈을 호출 할 때는 .py를 뺀 모듈 이름을 import 옆에 적어 준다.
- 호출 할 수 있는 함수는 사용자가 .py 파일 내부에 작성한 함수들이다.

mod1.py ✕

mod1.py > ...

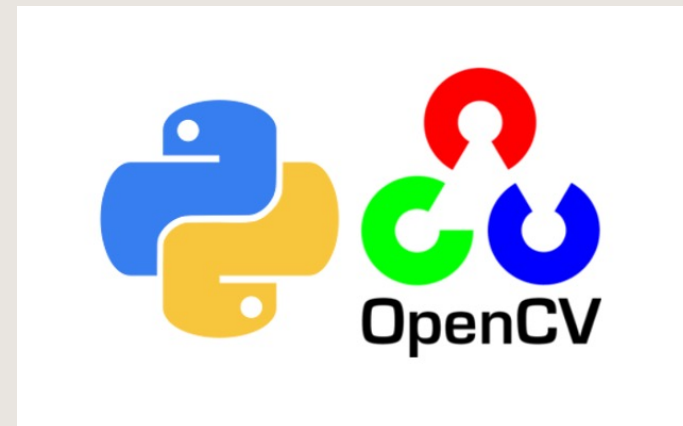
```
1  def sumAll(list):  
2      tmp = [n for n in list]  
3      return sum(tmp)
```

```
import mod1  
a = [1, 2, 3, 4, 5]  
print(mod1.sumAll(a)) 15
```

```
from mod1 import sumAll  
a = [1, 2, 3, 4, 5]  
print(sumAll(a)) 15
```

서드 파티 모듈(3rd Party module)

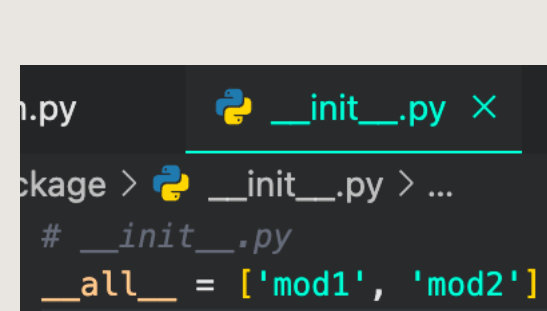
- 회사나 단체가 개발한 모듈들 이다.
- pip(파이썬 패키지 관리자)를 통해 설치 할 수 있다. (pip install 모듈 이름)
- 딥러닝, 머신러닝, 게임, 안드로이드, 데이터 분석 등을 지원하는 여러 라이브러리가 있다.



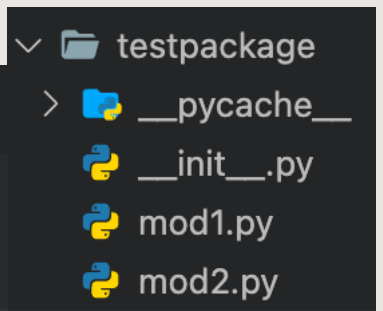
패키지(package)

- 모듈들을 모아놓은 것으로 폴더 형태로 나타낸다.
- 모듈을 주제별로 분류할 때 사용한다.
- `__init__.py`로 모듈들을 패키지의 일부임을 명시해준다.(PEP420 덕분에 python3.3부터는 없어도 인식)
- “from 패키지명.모듈명 import 함수명”으로 사용한다.

```
from testpackage.mod2 import avg
from testpackage.mod1 import add
add()
l1 = [1, 2, 3, 4, 5]
print(avg(l1))
```

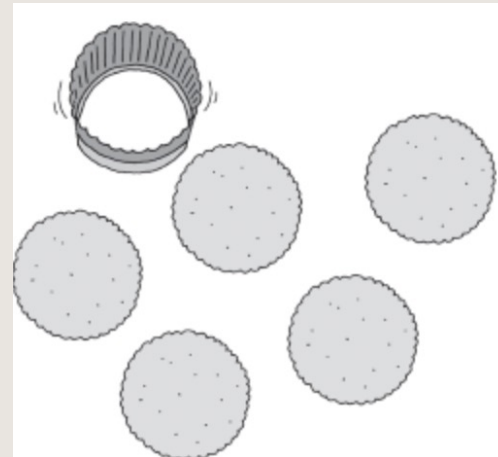


```
# __init__.py
__all__ = ['mod1', 'mod2']
```



class 클래스

- 클래스란 과자를 만드는 틀이라고 생각하면 편하다.
- 이러한 클래스로 만들어지는 것이 객체(Object) 이다.
- 객체는 객체 마다 고유한 성격을 가지고 동일한 클래스로 만든 객체들은 서로 전혀 영향을 주지 않는다.
- 클래스로 만든 객체를 인스턴스라고도 하는데 `a=Cookie()` 이렇게 만든 `a`는 객체이고 `a` 객체는 `Cookie`의 인스턴스이다. 즉, 인스턴스는 특정 객체가 어떤 클래스의 객체인지를 관계 위주로 설명할 때 사용된다.



클래스 개요

- C++과 모듈라3의 문법을 계승하여 간단한 클래스 정의 뿐만 아니라 연산자 Overloading, 연산자 Overwriting, 다중 상속 까지 지원한다.
- 클래스에 속한 변수를 **멤버**, 함수를 **메서드**라한다.
- 클래스의 첫번째 이름 대문자로 적는 것이 관행이다.

```
class Cookie:
    def __init__(self, a):
        self.a = a

    def re(self):
        return self.a
```

```
co1 = Cookie(10)
print(co1.a, co1.re())
```

1 1

```
class Cookie:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def re(self):
        return self.a+self.b+self.c
```

```
co1 = Cookie(10, 20, 30)
print(co1.a, co1.re())
```

10 60

클래스 - 생성자

- 생성자는 있어도 되고 없어도 됨.
- 객체를 초기화하는 용도로 사용
- `def __init__(self,인수1,...):`
 `self.멤버 명 = 인수1`
 ...

```
def __init__(self, a, b, c):  
    self.a = a  
    self.b = b  
    self.c = c
```

- 멤버 초기화 : “`self.멤버이름`” 형식으로 초기화 한다.
- 객체 생성 : “`객체 = 클래스명(인수)`”
- 인수를 초기값으로 전달하여 클래스 객체가 객체로 리턴 된다.
- `self`는 자기 자신을 가리키는 변수

클래스 - 상속

- 부모 클래스를 상속 받은 자식 클래스를 만들 수 있다.
- `super()` 메서드로 부모의 멤버를 호출 하여 자식 클래스에서 사용할 수 있다.
- Class 자식클래스명 (부모 클래스 명): 로 끝난다.

```
class Cookie:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def re(self):
        return self.a+self.b+self.c
```

```
class MilkCookie(Cookie):
    def __init__(self, a, b, c):
        super().__init__(a, b, c)

    def re(self):
        print("re")
        return super().re()
```

오버 라이딩, 오버 로딩

- 오버 라이딩 : 부모의 메서드를 자식 클래스에서 수정하는 행위이다. 메서드 명이 같아야 한다.
- 오버 로딩 : 같은 이름의 메서드 여러 개를 만들고 매개변수 유형과 개수가 다르도록 하는 기술이다.

```
def re(self):  
    return self.a+self.b+self.c
```

```
def re(self):  
    print("re")  
    return super().re()
```

```
def re(self):  
    return self.a+self.b+self.c
```

```
def re(self, a):  
    return a+self.b+self.c
```

```
def re(self, b):  
    return self.a+b+self.c
```

클래스의 멤버 보안

- private _ : 클래스 내부에서만 접근 가능하다.
- protected _ : 상속 된 클래스 및 클래스 내부에서만 접근 가능하다.
- public : 어디에서든지 접근 가능하다.(default)

```
class Cookie:
    def __init__(self, a, b, c):
        self.a = a
        self._b = b
        self.__c = c
```

```
1
Traceback (most recent call last):
  File "/Users/pupba/Desktop/work/...", line 1, in <module>
    print(co.c)
AttributeError: 'Cookie' object has no attribute 'c'
```

클래스의 특별한 메서드들

- `__del__()` : 인스턴스 소멸할 때 자동 실행(소멸자)
- `__repr__()` : 인스턴스를 `print()` 문으로 출력할 때 실행
- `__add__()` : + 연산을 한다.
- `__lt__()`, `__le__()`, `__gt__()`, `__ge__()`, `__eq__()`, `__ne__()` : 순서대로 왼쪽 부터 `<`, `<=`, `>`, `>=`, `==`, `!=` 연산을 한다.

```
class Cookie:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def __del__(self):
        print("bye")

    def __repr__(self):
        return str(self.a+self.b)
```

```
co = Cookie(1, 2, 3)
print(co)
del co
```