PYTHON PROGRAMMING

LECTURE 3: VARIABLE & OPERATOR

goorm

KAIST AI
Graduate School of AI

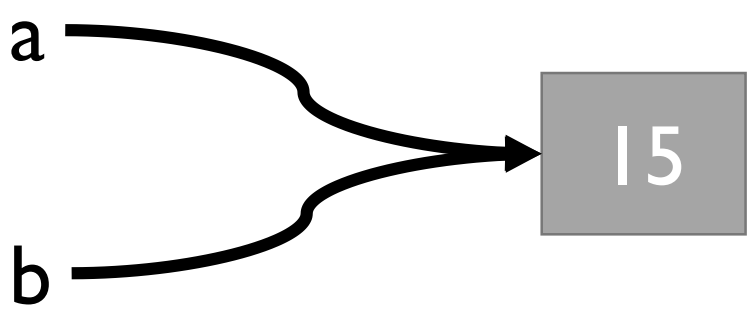
DAVIAN

Introduction to Variable

- 변수: 값을 저장하는 공간
  - = 연산자로 대입 연산
- 파이썬 변수의 특징
  - 모든 변수는 메모리 주소를 가르친다. (즉, 모든 것은 포인터다)
  - 변수 명 ← 일종의 이름표
    - 선언한 변수에 특정 공간이 생기는 개념이 아님
    - 필요하면 공간을 만들고 변수명을 붙이는 격

```
>>> a = 15
>>> b = 4
>>> a + b
19
```

```
>>> a = 15
>>> b = a
>>> b
15
```



How to Name Variables

- **알파벳, 숫자, 언더스코어(_)로 선언**
  - 숫자로 변수명을 시작할 수는 없음
  - Ex) data = 0, _a12 = 2, _gg = 'afdf'
- **변수명은 그 변수의 특징이 잘 살아 있게 하자 (가독성은 중요하다!)**
  - Ex) name = 'Hojun Cho'
- **변수명은 대소문자가 구분**
  - Ex) abc와 Abc는 다름
- **변수명으로 쓸 수 없는 예약어가 존재**
  - Ex) for, if, else, True 등

Assigning Variables

- C와 달리 대입 연산이 딱히 반환 값을 가지는 것은 아님

```
>>> (a = 2) == 2
    File "<stdin>", line 1
    (a = 2) == 2
       ^
SyntaxError: invalid syntax
```

- 연속해서 대입 가능 (C와 같이 뒤에서 부터 대입)

```
>>> a = b = 2
>>> a
2
>>> b
2
```

- (Python 3.9 이상) := 연산으로 대입과 동시에 반환 가능

```
>>> (a := 2) == 2
True
```

Primitive Data Types

원시 자료형: 가장 기본이 되는 자료형

유형			비고	예시
수치자료	정수	int	Overflow가 발생하지 않음 명시적인 Short type 등이 없음	1 2 3 100 -50 134251251523445 1234
	실수	float	부동소수점, Double precision	1.7 -5.7 3e5 .08 9. 4.67e-3
	복소수	complex	실수부와 허수부로 표현 허수부는 j 또는 J로 표현	1+8j 1.6+.8J
문자열		string	따옴표로 표현, Unicode 큰 따옴표, 작은 따옴표 차이 없음	'text' "한글도 되요" "a"
논리		bool	참/거짓을 표현 True, False 값 밖에 없음	True False
None		None	None 타입, 일종의 null None 값 밖에 없음	None

Example: Data Types

```
>>> a = 15      # 정수
>>> a
15
>>> b = 1.5     # 실수
>>> b
1.5
>>> c = 1+5j    # 복소수
>>> c
(1+5j)
>>> d = "text" # 문자열
>>> d
'text'
>>> e = True    # Boolean, 대소문자에 주의
>>> e
True
>>> f = None    # None, 대소문자에 주의
>>> f           # 아무것도 출력하지 않음
>>>
```

Arithmetic Operator

- 산술 연산을 위해서 산술 연산자를 활용

연산자	비고
+	덧셈
-	뺄셈
*	곱셈
**	거듭 제곱
/	나누기
//	나누기의 몫
%	나누기의 나머지

```
>>> 50 + 12
62
>>> 30 - 46
-16
>>> 2 * 6
12
>>> 2 ** 3
8
>>> 1.7 / 0.8
2.125
>>> 13 // 5
2
>>> 13 % 5
3
```

Bit Operator

- 비트 연산을 위해서 비트 연산자를 활용

연산자	비고
~	비트 부정
|	비트합
&	비트곱
^	배타적 비트합
>> <<	비트 시프트

```
>>> ~5
-6

>>> 2 | 3
3

>>> 2 & 3
2

>>> 2 ^ 3
1
```

Ex) $5 \rightarrow 0101_{(2)}$, $\sim 5 \rightarrow 1010_{(2)} \rightarrow -6$

Features of Arithmetic & Bit Operator

- 산술 연산자와 비트 연산자는 대입 연산자와 함께 축약 가능
  - 단, a += 1는 In-place 연산이고 a = a + 1은 Out-place 연산
  - Out-place 명시적으로 새로운 객체 생성
  - In-place 연산은 기존 객체를 수정 **시도**하고, 불가능할 시 새로운 객체 생성
    - 추후 자세히 설명

```
>>> a = 5

>>> a = a + 1
>>> a
6

>>> a += 1
>>> a
7
```

```
>>> a = 7

>>> a = a ^ 4
>>> a
3

>>> a ^= 4
>>> a
7
```

- a++ 연산은 파이썬에 존재하지 않음

Condition Operators

- 객체 간의 비교를 위해서 비교 연산자를 활용

비교연산자	비교상태	설명
x < y	~보다 작음	x가 y보다 작다
x > y	~보다 큼	x가 y보다 크다
x == y	같음	x와 y가 값이 같다
x is y		x와 y가 주소가 같다
x != y	같지 않음	x가 y가 값이 다르다
x is not y		x가 y가 주소가 다르다
x >= y	크거나 같음	x가 y 이상이다
x <= y	작거나 같음	x가 y 이하이다
x in X	포함	x가 X에 포함된다, $1 \text{ in } [1, 2, 3]$
x not in X	포함하지 않음	x가 X에 포함되지 않는다, $1 \text{ not in } [2, 4, 5]$

Condition Operator

- bool끼리의 연산을 위해서 논리 연산자를 활용
  - 비트 연산자와는 다르다!
  - Python에서의 비교연산은 한번에 평가된다
    - $(1 < 2 < 3) == (1 < 2 \text{ and } 2 < 3)$, $a == 4 > 3$

연산자	비고
not	부정
or	논리합
and	논리곱

```
>>> a = 3
>>> a < 4
True

>>> a > 2
True

>>> 2 < a and a < 4
True

>>> 2 < a < 4
True

>>> not a > 3
True
```

Operator Priority

연산자	설명
(...) [...] {...}	괄호, 리스트, 딕셔너리 생성
x[...] x(...) x.attr	인덱싱, 함수 호출, 속성 참조
**	거듭제곱
+x -x ~x	단항 연산자
* / // &	곱셈, 나눗셈 등
+ -	덧셈, 뺄셈
<< >>	비트 시프트
&	비트곱
^	배타적 비트합
|	비트합
in not in is is not < <= > >= == !=	포함 & 비교 연산자
not x	논리 부정
and	논리곱
or	논리합
[x] if [contion] else [y]	삼항 연산자 (추후 설명)

Features of Primitive Data Types

- 문자열 (str) 타입은 따옴표 또는 큰 따옴표로 정의
  - Ex) sentence = “hello”, sentence = ‘python’
  - 기능상 차이는 없으나 한 쪽을 쓰면 다른 쪽을 Text에 넣을 수 있음

```
>>> sentence = 'hello "python"'
>>> sentence
'hello "python"'
```

- **Primitive Data Type 들은 Immutable Type (불변 타입)이다.**
  - Python의 모든 것은 객체이기 때문에 Primitive Data Type 들 역시 객체
  - 그러나 불변타입들은 저장된 값이 변하지 않는다!
  - 모든 타입은 Physical Memory 주소를 가르침 (C에서의 pointer)
  - Primitive Data Type과 Tuple을 제외한 다른 모든 파이썬 객체는 Mutable Type (가변 타입)

Immutable Types & Mutable Types

- 파이썬에서 대입은 원칙적으로 메모리 주소 복사 **(즉, 값을 복사하지 않고 같은 주소를 공유)**
  - 불변형의 경우 수정이 필요할 경우에 **새로운 객체를 생성**

```
>>> a = 10                                  # int는 불변 타입
>>> b = a                                   # a와 b는 같은 메모리 주소를 가르침
>>> a += 1                                  # a는 불변 타입 -> 객체를 새로 생성해서 할당 (a = a + 1)
>>> a, b, a is b                            # a가 새로 할당되었기 때문에 a is not b
(11, 10, False)

>>> a = [1, 2, 3]                           # List는 가변 타입
>>> b = a                                   # a와 b는 같은 메모리 주소를 가르침
>>> a += [4]                                # a는 가변 타입 -> 원 객체를 수정
>>> a, b, a is b                            # a의 메모리 주소는 변함 없으므로 a is b
([1, 2, 3, 4], [1, 2, 3, 4], True)

>>> a = a + [5]                             # a에 [5]를 추가한 객체를 생성해서 a에 할당
>>> a, b, a is b                            # a가 새로 할당되었기 때문에 a is not b
([1, 2, 3, 4, 5], [1, 2, 3, 4], False)
```

Features of Data Assignment

- 파이썬에서 적당한 크기의 Primitive Data 대입은 **기존 객체를 할당** (불변 타입이라 상관 없음)

```
>>> a = 1                                   # int는 Primitive Type
>>> b = 1                                   # b에 새로 할당한 것 처럼 보이지만 기존 객체를 가져옴
>>> a is b                                  # a와 b는 같은 메모리 주소를 가지는 지 확인
True

>>> a = 13453436                            # int는 Primitive Type이지만 복잡한 값을 가짐
>>> b = 13453436                            # b에 값을 새로 할당
>>> a is b                                  # a와 b는 같은 메모리 주소를 가지는 지 확인
False

>>> a = 'text'                              # bool는 Primitive Type, 짧은 텍스트
>>> b = 'long-long-text'                    # bool는 Primitive Type, 긴 텍스트
>>> a is 'text', a == 'text', b is 'long-long-text', b == 'long-long-text'
(True, True, False, True)

>>> a = True                                # bool는 Primitive Type, True/False 모두 간단
>>> a is True
True

>>> a = None                                # None type은 Primitive Type & None 값만 존재
>>> a is None
True
```

Dynamic Typing

코드 실행 지점에서 데이터의 Type을 결정함

```c
int first = 10
int second = 20

printf("%d", first + second);
```

C

```python
first = 10
second = 20

print(first + second)
```

Python

Implicit Type Conversion

```
>>> a = True

>>> a = a + 2
>>> a
3

>>> a = a + 1.5
>>> a
4.5

>>> a = a + 7.1j
>>> a
(4.5+7.1j)
```

```
>>> 4 / 3
1.3333333333333333

>>> 4 // 3
1
```

- bool → int → float → complex
- None 타입과, str 타입은 별개
- int간의 나누기 → float (정수 나누기는 //)

```
>>> a = 1

>>> a + None
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'

>>> a + 'text'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Explicit Type Conversion

```
>>> a = 12345
>>> a
12345

>>> float(a)
12345.0

>>> complex(a)
(12345+0j)

>>> str(a)
'12345'

>>> bool(a)
True
```

```
>>> int(75.75)
75

>>> str(75.75)
'75.75'

>>> int(True)
1

>>> int(False)
0

>>> bool(None)
False
```

```
>>> float('75.75')
75.75

>>> bool('False')
True

>>> bool('')
False
```

- [*Type*]([*value*])로 명시적 형 변환 가능
  - int(a), float(text), str(value)
- 적절한 text는 적절한 값으로 변형
- 실수 → 정수: 소수점 내림
  - 반올림의 경우 *round* 내장함수 사용
- 빈문자열, 0, None 은 False로 변환

Type Checking

```
>>> a = 123
>>> b = 12.3
>>> c = '12.3'

>>> type(a)
<class 'int'>
>>> type(b)
<class 'float'>
>>> type(c)
<class 'str'>
```

- type 함수로 변수의 타입 확인 가능
- isintance 함수로 변수가 지정 타입인지 확인
  - isinstance([variable], [type])

```
>>> isinstance(a, float)
False
>>> isinstance(b, float)
True
>>> isinstance(c, str)
True
```

PYTHON PROGRAMMING

LECTURE 4: DATA STRUCTURE

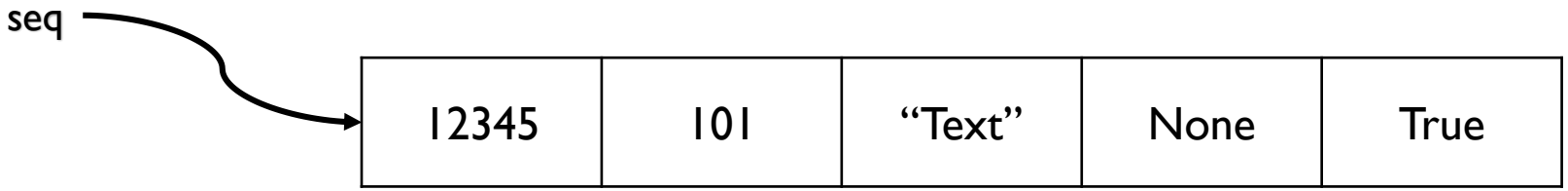
goorm

KAIST AI
Graduate School of AI

DAVIAN

List

- **배열: 일련의 데이터를 하나로 묶음**



- **파이썬 배열의 특징**
  - 대괄호로 선언 “ [value1, value2, …] ”
  - 아무 타입이나 넣기 가능
  - 길이가 정해져 있지 않음

```
>>> seq = [12345, 101, "Text", None, True]
>>> seq
[12345, 101, 'Text', None, True]
```

List Index & Slicing

```
>>> seq = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> seq[0]       # 첫번째 요소
1
>>> seq[8]
9
>>> seq[-1]      # 뒤에서 첫번째 요소
10
>>> seq[-3]
8
```

- **List indexing**
  - seq[index] 형태로 요소 하나 접근
  - 0부터 숫자세기 시작
  - 음수 가능 (뒤에서 부터 접근)

```
>>> seq[:3]              # 앞 3번째 전까지
[1, 2, 3]
>>> seq[3:]              # 앞 3번째 부터
[4, 5, 6, 7, 8, 9, 10]

>>> seq[3:-1]            # 앞 3번째 부터 뒤에 1번째 전까지
[4, 5, 6, 7, 8, 9]
>>> seq[-3:-1]
[8, 9]

>>> seq[::2]             # 두 칸 씩 뛰면서
[1, 3, 5, 7, 9]
>>> seq[9:2:-1]          # 한 칸 씩 뒤로 가면서
[10, 9, 8, 7, 6, 5, 4]
>>> seq[-2:2:-1]
[9, 8, 7, 6, 5, 4]
```

- **List Slicing**
  - seq[start: end: step] 형태로 List 자르기
  - end번째는 포함하지 않음

List Operators

```
>>> a = [1, 2, 3, 4]
>>> b = [5, 6, 7, 8]
>>> a + b                          # 리스트 합치기
[1, 2, 3, 4, 5, 6, 7, 8]

>>> a[0] = 'something?'            # 리스트 내에 값 바꾸기

>>> a * 2                          # 곱하기로 여러 개를 여러 번 합치기
['something?', 2, 3, 4, 'something?', 2, 3, 4]

>>> 'something?' in a              # 리스트 안에 요소가 있는지 확인
True
```

List Operators

```
>>> seq = [1, 2, None, True]
>>> len(seq)          # 리스트 길이 구하기, 내장 함수 형태, seq.length가 아니다!
4

>>> seq.append("something")       # 맨 뒤에 요소 추가, 메소드 형태
>>> seq.extend([5, 6])            # 맨 뒤에 리스트 추가, seq += [5, 6]
>>> seq.insert(1, 1.5)            # 원하는 곳에 삽입, 1번째에 1.5 삽입, (index, val)
>>> seq
[1, 1.5, 2, None, True, 'something', 5, 6]

>>> del seq[1]                    # 1번째 요소 삭제, 예약어 형태, seq.delet가 아니다!
>>> seq.remove('something')       # 원하는 값을 삭제
>>> seq
[1, 2, None, True, 5, 6]
```

Reserved words & Built-in Functions & Methods

파이썬에서 제공되는 기능은 일반적으로 다음과 같은 형태

예약어

- 일종의 문법적인 요소
- 괄호를 쓰지 않음
- 재정의 불가능

- 예시
  - del
  - if … else …
  - assert

내장함수

- 기본 정의된 함수
- 별개의 함수 사용
- 재정의 가능
- 편의성 향상

- 예시
  - len( )
  - sum( )
  - range( )

메소드

- 객체 내에 정의된 함수
- .method( ) 으로 접근
- Overriding
- 해당 객체를 다룸

- 예시
  - .append( )
  - .insert( )
  - .extend( )

Reserved words

Python Keywords

False	def	if	raise
None	del	import	return
True	elif	in	try
and	else	is	while
as	except	lambda	with
assert	finally	nonlocal	yield
break	for	not	
class	from	or	
continue	global	pass	

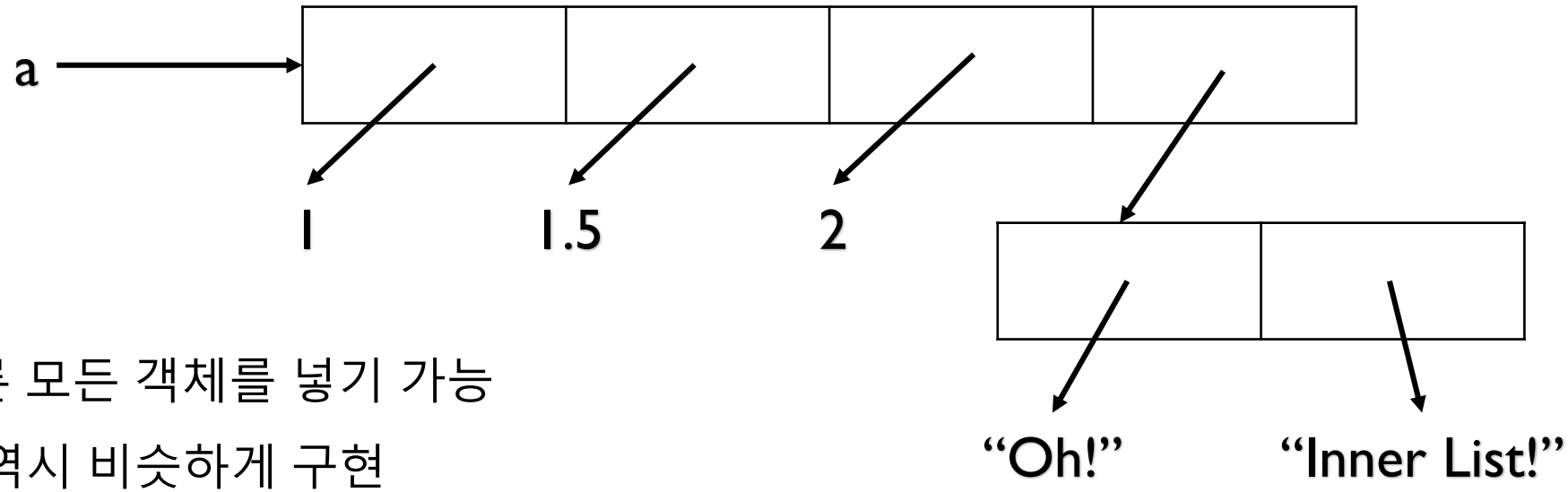
Built-in Functions

		Built-in Functions		
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()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Memory Structure of List

```
>>> a = [1, 1.5, 2, ["Oh!", "Inner List!"]]          # List in list
>>> a
[1, 1.5, 2, ['Oh!', 'Inner List!']]

>>> a = [[1, 2, 3], [4, 5, 6]]                       # 이차원 리스트
>>> a
[[1, 2, 3], [4, 5, 6]]
```



- 리스트 안에 다른 모든 객체를 넣기 가능
  - 2차원 배열 역시 비슷하게 구현

List as Mutable Data Type

리스트 등은 가변 타입임으로 주의!

```
>>> a = [1, 2, 3, 4, 5]      # 가변 타입
>>> b = a                    # 같은 주소를 가르침
>>> a += [6]                 # In-place 연산, a = a + [6] 과 다름
>>> b
[1, 2, 3, 4, 5, 6]

>>> a = ‘Something’          # 불변 타입
>>> b = a                    # 같은 주소를 가르침
>>> a += '?'                 # 불변 타입이기에 Out-place, a = a + ‘?’
>>> b
'Something'
>>> a
'Something?'
```

Time Complexity of List

Operation	Example	Big-O	Notes
index	l[i]	*O(1)*	
store	l[i] = 0	*O(1)*	
length	len(l)	*O(1)*	
append	l.append(5)	*O(1)*	
pop	l.pop()	*O(1)*	l.pop(-1) 과 동일
clear	l.clear()	*O(1)*	l = [] 과 유사
slice	l[a:b]	*O(b-a)*	l[:] : O(len(l)-0) = O(N)
extend	l.extend(…)	*O(len(...))*	확장 길이에 따라
construction	list(…)	*O(len(...))*	요소 길이에 따라
check ==, !=	l1 == l2	*O(N)*	비교
insert	l .insert(i, v)	*O(N)*	i 위치에 v를 추가
delete	del l[i]	*O(N)*	
remove	l.remove(…)	*O(N)*	
containment	x in/not in l	*O(N)*	검색
copy	l.copy()	*O(N)*	l[:] 과 동일 - O(N)
pop	l.pop(i)	*O(N)*	l.pop(0):O(N)
extreme value	min(l)/max(l)	*O(N)*	검색
reverse	l.reverse()	*O(N)*	그대로 반대로
iteration	for v in l:	*O(N)*	
sort	l.sort()	*O(N Log N)*	
multiply	k*l	*O(k N)*	[1,2,3] * 3 » O(N**2)

Python List는
동적 배열로
구현됨

Tuple

- **불변 타입 리스트 (Immutable List)**
- 선언 시 "[ ]" 가 아닌 "( )"를 사용, 문맥에 따라 괄호 생략 가능
- 리스트의 연산, 인덱싱, 슬라이싱 등을 동일하게 사용

```
>>> t = (1, 2, 3, 4)                    # 선언
>>> t = 1, 2, 3, 4                      # 괄호 생략가능
>>> t
(1, 2, 3, 4)

>>> len(t)                              # List 메소드 함수들 사용 가능
4
>>> t * 2
(1, 2, 3, 4, 1, 2, 3, 4)

>>> t[3] = 5                            # 불변 타입이라 수정은 불가능
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Features of Tuple

- 일반적으로 함수에서 2개 이상 요소를 반환할 때 사용
- Tuple은 불변 타입 이지만 Tuple안의 요소는 가변 타입일 수도 있음
- 문자열 타입 (str)의 경우 일종의 문자 tuple로 생각 가능

  → Indexing 및 slicing이 가능

```
>>> (1)          # 그냥 괄호는 산술연산자
1
>>> (1,)         # 요소 하나를 가진 Tuple을 위해서는 ","를 적어야 함
(1,)
```

```
>>> a = (1, 2, [5, 6, 7])        # 불변 타입 안의 가변 타입
>>> a[2].append(8)                # 가변 타입 내 삽입은 문제 없음
>>> a
(1, 2, [5, 6, 7, 8])
```

```
>>> a = 'some text'       # 불변 타입
>>> a[:4]                 # Tuple의 연산자 사용가능
'some'
```

Packing & Unpacking

- 패킹: 여러 데이터를 묶기
  - Ex) t = 1, 2, 3, 4, 5
- 언패킹: 묶인 데이터를 풀기
  - Ex) a, b, c, d, e = t
    - * (Asterisk)로 남는 요소를 리스트로 남기기 가능

```
>>> t = [1, 2, 3, 4, 5]
>>> a, b, c, _, _ = t   # Unpacking
>>> c                   # "_"는 관습적으로 사용하지 않는 변수에 사용
3

>>> a, *b, c = t        # * 로 남는 부분을 리스트로 묶을 수 있음
>>> a, b, c
(1, [2, 3, 4], 5)
```

Dictionary

- 일종의 매핑을 위한 데이터 구조
  - Key → Value 형태로 구현
  - 불변 타입으로만 이루어져 있으면 Key로 사용 가능
  - {Key1: Value1, Key2: Value2, Key3: Value3, …} 형태로 선언

Key → Value
1 → 'something'
(1, 2.5) → 1.5
'text' → 2

```
>>> dictionary = {
...     1: 'something',          # 1을 'something'에 매핑
...     (1, 2.5): 1.5,           # 정확히는 Hashable할 경우 Key로 사용 가능
...     'text': 2,               # 마지막 콤마는 무시
... }

>>> dictionary[1]               # 대괄호로 indexing
'something'

>>> dictionary['text']
2

>>> dictionary[1, 2.5]          # Tuple이므로 괄호 생략 가능
1.5
```

Addition & Deletion on Dictionary

```
>>> dictionary = {}                    # 빈 딕셔너리 생성, dict()로도 가능

>>> dictionary['text'] = 1             # 요소 삽입
>>> dictionary['list'] = [5, 6, 7]     # Value는 아무 객체나 가능
>>> dictionary
{'text': 1, 'list': [5, 6, 7]}

>>> dictionary['text'] = 'Oh?'         # 키는 중복이 불가능 → 덮어 씌워짐
>>> dictionary
{'text': 'Oh?', 'list': [5, 6, 7]}

>>> del dictionary['text']             # 요소 삭제
>>> dictionary
{'list': [5, 6, 7]}

>>> len(dictionary)                    # 크기 확인
1
```

Listing Dictionary Items

```
>>> dictionary = {'한국어': 0, '영어': 1, '중국어': 2}
>>> dictionary
{'한국어': 0, '영어': 1, '중국어': 2}

>>> dictionary.items()              # 모든 key와 value를 일종의 Tuple List로 반환
dict_items([('한국어', 0), ('영어', 1), ('중국어', 2)])

>>> dictionary.keys()               # 모든 Key를 일종의 List로 반환
dict_keys(['한국어', '영어', '중국어'])
>>> dictionary.values()             # 모든 Value를 일종의 List로 반환
dict_values([0, 1, 2])

>>> 2 in dictionary.values()        # Value 안에 2가 있는지 검사
True
```

Time complexity of Dictionary

Operation	Example	Big-O	Notes
Index	d[k]	O(1)	
Store	d[k] = v	O(1)	
Length	len(d)	O(1)	
Delete	del d[k]	O(1)	
Clear	d.clear()	O(1)	s = {} or = dict() 유사
Construction	dict(...)	O(len(...))	
Iteration	for k in d:	O(N)	

Dictionary는 Hash 로 구현: indexing 속도가 O(1)

Set

- Dictionary의 Key만 모여 있는 형태 → 집합형

```
>>> s = set([1, 2, 3, 'text'])          # Set 선언
>>> s                                   # 빈 Set은 set()으로 선언, {} 은 빈 Dictionary
{1, 2, 3, 'text'}

>>> s.add(4)                            # 요소 추가
>>> s.add('text')                       # 중복된 요소는 추가하지 않음
>>> s
{1, 2, 3, 4, 'text'}

>>> s.remove(2)                         # 요소 삭제
>>> s.remove(99)                        # 존재하지 않는 요소는 에러 발생
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 99

>>> s.discard(99)                       # 요소 삭제, 존재하지 않을 경우 무시
>>> s
{1, 3, 4, 'text'}

>>> s.update([1, 99, None, True])       # 여러 요소 추가, 중복은 무시
>>> s
{1, None, 3, 4, 99, 'text'}
>>> s.clear()                           # Set 비우기
>>> s
set()
```

Set Operations

- 수학적 집합 연산자가 존재

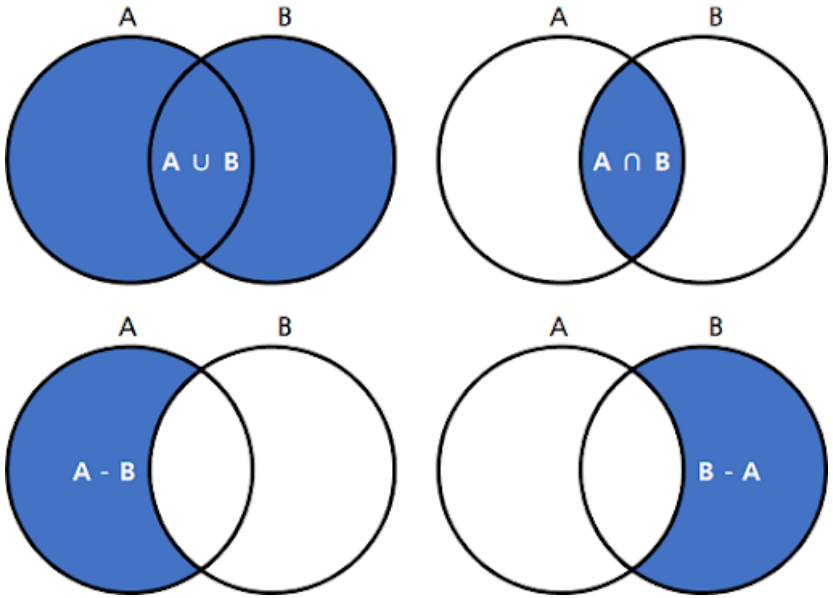
```
>>> s1 = set([1, 2, 3, 4])
>>> s2 = set([3, 4, 5, 6])

>>> s1 & s2                    # 교집합
{3, 4}

>>> s1 | s2                    # 합집합
{1, 2, 3, 4, 5, 6}

>>> s1 - s2                    # 차집합
{1, 2}

>>> s1 ^ s2                    # 배타적 합집합
{1, 2, 5, 6}
```



PYTHON PROGRAMMING

LECTURE 5: CONDITION & LOOP

goorm

KAIST AI
Graduate School of AI

DAVIAN

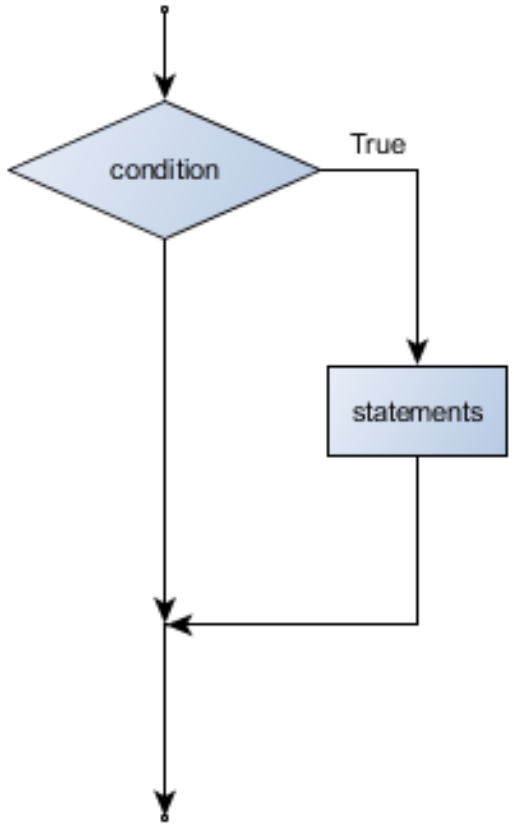
Conditional Statements

- 특정 조건이 만족될 경우 실행할 문항을 설정

```
명령 1
명령 2

if 〈조건〉:
        if-명령 1
        if-명령 2

명령 3
명령 4
```



- 들여쓰기와 :으로 구문을 구분
- 들여쓰기의 Convention은 스페이스 4칸
  - Tab 키를 눌러 삽입

Conditional Statements

- if [조건] – 조건을 검사하여 block을 실행
- elif [조건] – 이전 조건과 맞지 않을 경우 조건을 다시 검사 및 실행
- else – 이전 모든 조건이 맞지 않을 경우 실행

```
score = 80

if score > 60:
    print ("Over 60")

if score > 70:
    print ("Over 70")

if score > 80:
    print ("Over 80")
```

```
score = 80

if score > 80:
    print ("Grade A")

elif score > 70:
    print ("Grade B")

else:
    print ("Grade F")
```

Conditional Statements

```
if False:
    print("This sentence does not show")

if "":
    print("Empty is False")

if 0:
    print("0 is False")

if None:
    print("None is also False")
```

Conditional Statements

```
a, b = 5, 8

if a == 5 and b == 6:
    print ("This is False")

if not a < b < 9:
    print ("This is False")

if a + 3 == b:
    print ("This is True")
```

비교 연산자와
논리 연산자를
사용

Ternary operators

삼항 연산자

- [Value1] if [Condition] else [Value2]
- Condition이 참이면 Value1을 거짓이면 Value2를 반환
- **연산자이다**
  - 블록으로 구분되는 문법요소가 아님

```
>>> value = 32

>>> "odd" if value % 2 else "even"
'even'

>>> ("odd" if value % 2 else "even") + "_number"
'even_number'
```

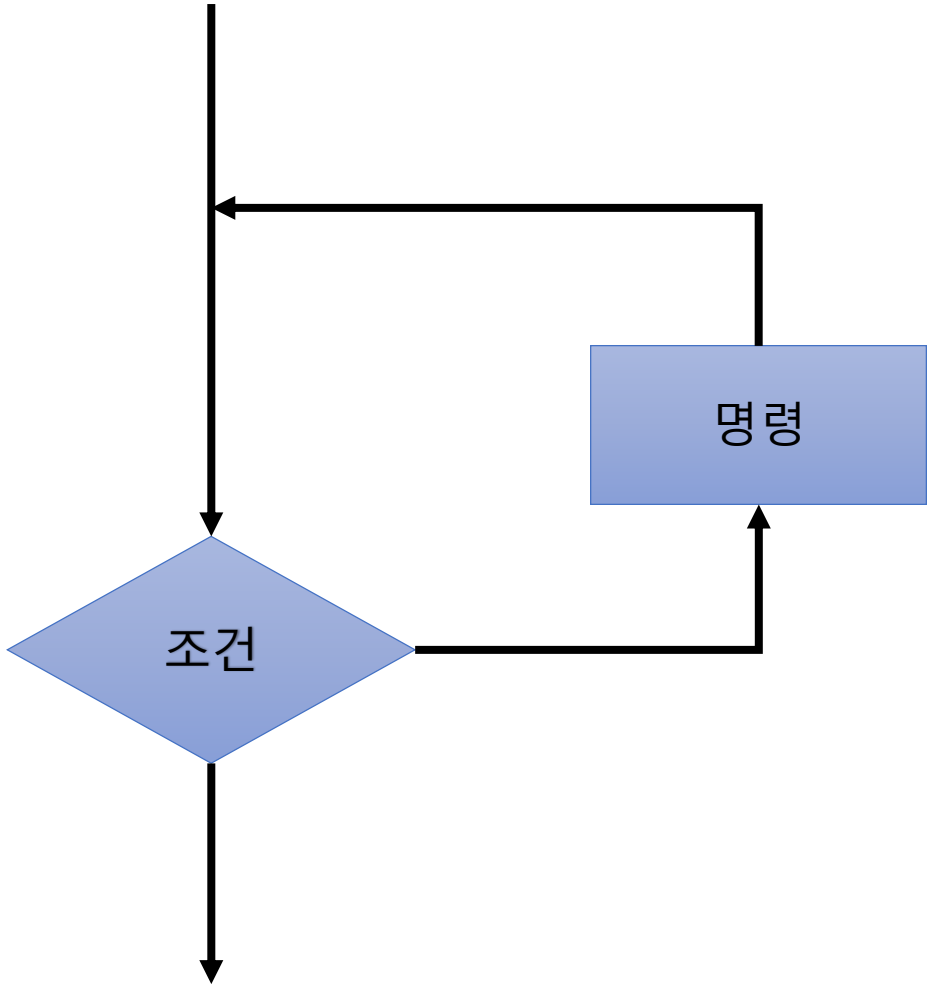
Loop

- 반복해서 구문을 수행

```
명령 1
명령 2

while 〈조건〉:
        while-명령 1
        while-명령 2

명령 3
명령 4
```



- 들여쓰기와 :으로 구문을 구분

While Statement

조건을 만족할 때까지 출력

```
i = 1                       # 초기 값

while i < 10:               # i가 10보다 작을 때까지
    print (i)               # i 출력
    i += 1                  # i에 1씩 더함
```

For Statement

- Python의 For문은 주어진 객체를 순환하는 개념
- for [Element] in [Iterable] 의 형태로 사용

```
for i in [0, 1, 2, 3, 4]:          # i가 리스트를 순환
    print (i)

for i in [0, 1, 2, 3, 4]:
    if i % 2:
        print(i, "is odd")
```

For Statement

```
for i in [1, 2, 3, 4, 5]:   # List를 순환
    print ("Test1", i)

for i in range (5):         # 0부터 5미만까지 순환
    print ("Test2", i)

for i in range (1, 6):      # 1부터 6미만까지 순환
    print ("Test3", i)

for i in range (1, 10, 2): # 1부터 10미만까지 2칸씩 뛰며 순환
    print ("Test4", i)
```

- **range** 내장 함수로 숫자 반복 생성 가능 (Generator 반환)
  - range (start, end, step) 형태로 사용, **End는 미포함**
- Generator: 리스트와는 다르게 숫자를 하나씩 생성 반환 (메모리 효율적)

Notions for Loop

- 반복문의 변수명은 일반적으로 i, j, k로 지정
- 반복문을 포함해서 프로그래밍에서 숫자의 시작은 대부분 0부터
- 반복문이 끝나지 않는 무한 loop에 주의
- 모든 순환이 가능한 객체는 for문을 적용하는 것이 가능

```
for c in "This is text":
    print (c)

for word in ["한국어", "문장", "처리"]
    print (word)

for key in {"text": 1, "word": 2}:
    print (key)
```

Controlling Loop – break & continue

Break문으로 가장 바깥의 반복문을 나가는 것이 가능

```
for i in range(100):
    if i % 17 == 0:
        break           # i가 17보다 커지면 반복 나가기
    print (i)
```

Continue문으로 가장 바깥의 반복문의 처음으로 되돌아가기 가능

```
for i in range(100):
    if i % 17:
        continue        # i가 17의 약수가 아니면 for 처음으로
    print (i)
```

Controlling Loop – else

Else문으로 반복을 완전이 돌았을 경우 실행되는 block 지정가능

```
for i in range(10):
    print (i)
else:
    print ("Loop complete with break")
```

Break로 중간에 나오게 되면 Else문이 실행되지 않음

```
for i in range(10):
    print (i)
    if i > 5:
        break
else:
    print ("Loop complete without break")
```

PYTHON PROGRAMMING

LECTURE 6: FUNCTION

goorm

KAIST AI
Graduate School of AI

DAVIAN

Function Definition

- 함수: 명령을 수행하는 일종의 기능 단위
  - 코드를 논리적으로 분리, 캡슐화 용도
  - 필요한 경우 반복적으로 호출
  - Return 명령어가 없는 경우 None을 반환

```
def [function]([parameter 1], [parameter 2], ...) :
        function-명령 1
        function-명령 2
        return [value]

명령 1
명령 2
value = function(argument 1, argument 2)
명령 3
명령 4
```

Function Definition Example

사각형의 넓이를 구하는 함수

```
def rectangle_area(x, y):
    return x * y

row = 10
col = 100
print(rectangle_area(row, col))
print(rectangle_area(20, 1.5))
```

Notion for Function Parameter

인자로 받은 값을 바로 수정하는 것은 권장하지 않음

```python
def function(seq):
    seq += [1]

seq = [1, 2, 3, 4, 5]
function(seq)
print(seq)                  # [1, 2, 3, 4, 5, 1]
```

```python
def function(seq):
    seq = list(seq)
    seq += [1]

seq = [1, 2, 3, 4, 5]
function(seq)
print(seq)                  # [1, 2, 3, 4, 5]
```

Variable Scope

- 파이썬에서는 상위에 정의된 변수는 언제나 참조 가능
- 함수내에 정의된 변수 이름은 그 함수 내에서만 유효

```
var1 = 10
var2 = 20

def function(var2):                 # var2를 이 함수 내에서 재정의
    var2 += 1                       # 기존 var2와 다름
    print (var1 + var2)             # 상위 변수 var1 읽기 가능

function(var2)
print (var2)
```

Variable Scope

```
var1 = 1                    # Global Variable

def main():                 # Global Function
    var2 = 10               # Local Variable

    def function():         # Local Function
        var3 = 100          # Local Variable
        print(var1, var2, var3)

    function()
    print(var1, var2)

main()
print(var1)
```

Global

- 최상위에 선언
- 다른 파일에서 접근 가능

Local

- 함수 안에 선언
- 상위 함수에서는 접근 불가

Global & Nonlocal

```
var = 1

def main():
    var = 10

    def function1():
        global var    # 최상위 변수 재정의 선언
        var += 1

    def function2():
        nonlocal var  # 직상위 변수 재정의 선언
        var += 1

    function1()
    function2()
    print(var)

main()
print(var)
```

- 상위 변수 재정의 선언
- 함수 맨 앞에 선언함
  - **global**: 최상위 변수
  - **nonlocal**: 바로 상위 변수

- 스파게티 코드의 주 원인
  - 사용하지 않는 것을 권장

Variable Capture?

```
var = 1

def function():
    print (var)

var += 1
function()  # 2
```

- 함수형 언어에서 쓰이는 Capture와 다름
- 상위 값이 바뀌면 하위 값이 바뀜
- **즉, closure가 아님**

- **프로그램을 스파게티로 만드는 주요 원인**
  1. 상위 객체엔 가능하면 접근하지 않기
  2. 되는대로 모두 파라미터로 받자
  3. 최상위 선언도 가급적 지양

How to Make Closure?

```
number = 10

def print_closure_factory(number):          # Closure Function Factory
    def print_closure():                    # Closure 함수
        print(number)                       # Factory의 변수를 사용
                                            # → Closer 마다 고유한 변수
    return print_closure                    # 만들어진 Closure를 반환

print_5 = print_closure_factory(5)
print_10 = print_closure_factory(10)

number += 10
print_5()
print_10()
```

- 파이썬에서 Closure는 Factory 형식으로 사용
- 파이썬에서는 함수도 일반 객체(일급 객체)이다
  - 변수로 할당 가능 → Argument & Return 가능

Closure Example

```
def add(var):
    return var + 2

def multiply(var):
    return var * 2

def factory(function, n):                    # 함수를 파라미터로 받는 Factory
    def closure(var):                        # Closure 생성
        for _ in range(n):
            var = function(var)
        return var

    return closure                           # Closure를 반환

print(factory(add, 4)(10))
print(factory(multiply, 4)(3))
```

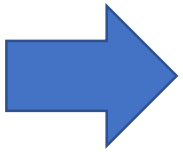
Decorator

```python
def print_closure_factory(function):
    def print_closure(var):
        print("Input:", var)
        out = function(var)
        print("Output:", out)

    return print_closure

def add(var):
    return var + 2

print_add = print_closure_factory(add)
print_add(10)
```



```python
def print_decorator(function):
    def print_closure(var):
        print("Input:", var)
        out = function(var)
        print("Output:", out)

    return print_closure

@print_decorator
def add(var):
    return var + 2

add(10)
```

꾸밈자 (Decorator)

- 함수 하나를 인자로 받아 같은 형태의 함수를 반환하는 함수
- @을 사용하여 함수를 꾸미는데 사용 가능
- Class를 사용할 시 Decorator에 인자 추가가 가능

Decorator with Argument

```
def times_decorator_factory(times):      # 인자를 받아 Decorator를 만듦
    def times_decorator(function):       # 함수를 받아 꾸밈
        def closure(var):                # 꾸며진 함수
            for _ in range(times):
                var = function(var)
            return var
        return closure
    return times_decorator

@times_decorator_factory(5)              # Decorator에 인자 추가
def add(number):
    return number + 2

print(add(5))
```

- Decorator에 인자를 추가하기 위해선 함수를 한번 더 wrapping 필요

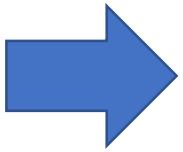
Appropriate Decorating

```
def print_decorator(function):
    def print_closure(var):
        print("Input:", var)
        out = function(var)
        print("Output:", out)

    return print_closure

@print_decorator
def add(var):
    return var + 2

print(add.__name__)
```



```
from functools import wraps

def print_decorator(function):
    @wraps(function)
    def print_closure(var):
        print("Input:", var)
        out = function(var)
        print("Output:", out)

    return print_closure

@print_decorator
def add(var):
    return var + 2

print(add.__name__)
```

- 함수를 wrapping하기 때문에 기존 함수에 접근 불가
  - Docstring, 함수 이름 등 기존 함수의 특성을 가져올 필요가 있음
  - functools 라이브러리의 wraps 데코레이터 사용

Recursive Function

재귀함수

- 자기 자신을 호출하여 반복적으로 수행
- 수학의 점화식과 동일
- 재귀 함수와 반복문은 수학적으로 동치 (서로 변환 가능)

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

print (factorial(5))
```

Function Parameters

- 인자를 명시적으로 대입 가능

```
def function(var1, var2):
    print(var1, var2)

function(var2=10, var1=20)       # (20, 10)
```

- 인자 기본값을 설정 가능

```
def function(var1, var2=20):
    print(var1, var2)

function(10)              # (10, 20)
function(10, 15)          # (10, 15)
```

- 기본값이 설정된 인자는 맨 뒤에 붙여서 써야만 함

```
>>> def function(var1, var2=20, var3):
...     print(var1, var2, var3)
...
File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
```

Variable Length Parameter

- 인자 개수가 정해져 있지 않다면...?
- * (Asterisk)를 사용하여 남은 여러 인자를 Packing 가능
- 가변인자는 맨 마지막에 단 한 개만 위치 가능

```
def add_all(a, b, *args):          # 관습적으로 가변인자는 args를 사용
    print(args)                    # (3, 4, 5)

    sum = 0
    for elem in args:
        sum += elem

    return a + b + sum

print(add_all(1, 2, 3, 4, 5))    # 15
```

Keyword Variable Length Parameter

- 만약 명시적으로 지정된 파라미터가 남는다면? → 키워드 가변인자
- ** (Double asterisk)를 사용하여 남는 키워드 변수를 packing
- Dictionary 형태로 반환

```
def print_args(a, *args, **kwargs):       # 키워드 가변인자는 관습적으로 kwargs 사용
    print(args, kwargs)                   # (2, 3) {‘var1’: 100, ‘var2’: 200}

print(print_args(1, 2, 3, var1=100, var2=200))
```

- 파라미터 순서: 일반 인자 → 기본값 인자 → 가변 인자 → 키워드 가변인자

```
def function(var1, var2=10, *args, **kwargs):
    print(var1, var2, args, kwargs)               # 1 2 (3,) {'var3': 10}

function(1, 2, 3, var3=10)
```

Parameter Unpacking

- Seqeunce에 * 을 붙이면 Unpacking
  - 리스트, 튜플에 적용가능

```
def function(a, b, c):
    print(a, b, c)

l = [1, 2, 3]
function(*l)          # 1 2 3
```

- Dictionary에 ** 을 붙이면 Keyword unpacking

```
def function(var1, var2, **kwargs):
    print (var1, var2, kwargs)

d = {
    'var1': 10,
    'var2': 20,
    'var3': 30
}
function (**d)          # 10 20 {'var3': 30}
```

Type hints

- 파이썬은 동적 타이핑 → 다소 interface를 알기 어려움
- 함수에 타입 힌트 제공이 가능
  - [function]([var]: [type], ...) 의 형태

```
# string과 integer를 받아 string을 반환
def multply_text(text: str, n: int) -> str:
    return text * n
```

- 그러나 딱히 타입을 안 맞춰도 에러가 안 남
  - Runtime에 타이핑을 하는 것은 동일

PYTHON PROGRAMMING

LECTURE 7: PYTHONIC PROGRAMMING

goorm

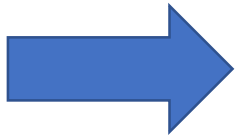
KAIST AI
Graduate School of AI

DAVIAN

Comprehension

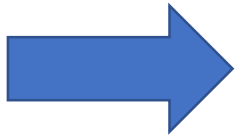
- List, Dictionary 등을 빠르게 만드는 기법
  - for + append 보다 속도 빠름

```
result = []
for i in range(10):
    result.append(i * 2)
```



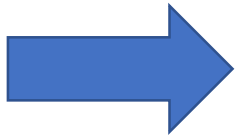
```
result = [i * 2 for i in range(10)]
```

```
result = {}
for i in range(10):
    result[str(i)] = i
```



```
result = {str(i): i for i in range(10)}
```

```
result = set()
for i in range(10):
    result.add(str(i))
```



```
result = {str(i) for i in range(10)}
```

Comprehension

- if 문을 마지막에 달아 원하는 요소만 추가 가능

```
evens = [i for i in range(100) if i % 2 == 0]
```

- 겹 for문 사용 가능

```
result = [(i, j) for i in range(5) for j in range(i)]
```

- 다차원 배열 만들기가 매우 유용

```
eye = [[int(i == j) for j in range(5)] for i in range(5)]
```

Generator

- **range** 함수의 경우 숫자를 하나씩 생성하여 반환
  - 이러한 요소를 하나씩 생성해서 반환하는 객체를 **Generator**라고 함

```
def my_range(stop):
    number = 0
    while number < stop:
        yield number
        number += 1

for i in my_range(5):
    print (i)
```

- function에 yield를 사용할 시 Generator가 됨
- yield 하는 위치에서 값을 반환
- 다시 값을 요청 받을 시 yield 다음 줄부터 실행
- Return 될 시 반복을 멈춤
  - 정확히는 StopIteration Error 발생
- Sequence 전체를 생성하는 것이 아니므로 메모리 효율적
  - 매우 큰 데이터 셋을 처리할 땐 Generator 사용 권장

- 괄호로 Generator Comprehension 형태로 선언 가능
  - Function 등으로 이미 괄호 쳐져 있다면 괄호 생략 가능

```
even_generator = (i * 2 for i in range(100))
```

Built-in Functions

- sum ([Iterable])

```
>>> sum([1, 2, 3, 4, 5])                    # 리스트 내의 합
15
>>> sum(i for i in range(1, 101) if i % 2 == 0) # 1에서 100까지 짝수 합
2550
```

- any ([Iterable]), all ([Iterable])

```
>>> any([False, True, False])               # 하나라도 참
True
>>> all([False, True, False])               # 모두 참
False
```

- max ([Iterable]), min ([Iterable])

```
>>> max([7, 5, -2, 5, 8])                   # 가장 큰 값
8
>>> min([7, 5, -2, 5, 8])                   # 가장 작은 값
-2
```

Zip

- 2 개 이상의 순환 가능한 객체를 앞에서 부터 한번에 접근할 때 사용

```
seq1 = ['What', 'is', 'zip']
seq2 = [True, False, True]
seq3 = [1, 2, 3, 4]                          # 길이가 안 맞을 경우 남는 건 버림

for w1, w2, w3 in zip(seq1, seq2, seq3): # 앞에서 부터 하나씩 빼어 Tuple로 반환
    print(w1, w2, w3)
```

- Unpacking을 이용하여 2차원 리스트의 열 단위 접근 역시 가능

```
array = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

for row in array:                    # 행 단위 접근
    print(row)

For col in zip(*array):              # 열 단위 접근
    print(col)
```

- **seq2 = zip(*seq1)의 역연산은 seq1 = zip(*seq2) 이다!**

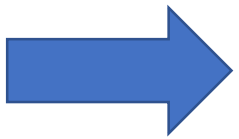
enumerate

- For문이 Sequence를 돌 때 그 index가 필요할 때가 있음

  → **enumerate** 내장 함수 사용

```
seq = ['This', 'is', 'sentence']

for i in range(len(seq)):
    print (i, seq[i])
```



```
seq = ['This', 'is', 'sentence']

for i, word in enumerate(seq):
    print (i, word)
```

- zip과 enumerate를 동시에 사용하는 등 여러 Generator를 한번에 사용

```
seq1 = ['This', 'sentence']
seq2 = [True, False]

for i, (a, b) in enumerate(zip(seq1, seq2)):
    print(i, a, b)
```

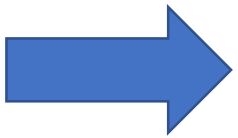
- Generator는 List형태로 출력하기 위해선 list로 변환 필요

```
>>> list(enumerate(['This', 'is', 'sentence']))
[(0, 'This'), (1, 'is'), (2, 'sentence')]
```

Lambda Function

- 함수의 이름 없이 빠르게 만들어 쓸 수 있는 익명 함수
- 수학에서의 람다 대수에서 유래

```
def add(a, b):
    return a + b
```



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

- lambda [param1], [param2], ...: [expression] 형태로 사용

- 여러 줄을 쓸 수 없음
- 공식적으로는 Lambda의 사용을 권장하지 않음, 그러나 많이 씀
  - 문서화 지원 미비
  - 이름이 존재하지 않는 함수가 생성
  - 복잡한 함수 lambda로 작성할 시 가독성 하락

Built-in function

- map ([function], [iterable])
  - 각 요소에 function 함수를 적용하여 반환

```
>>> seq = [6, -2, 8, 4, -5]

>>> list(map(lambda x: x * 2, seq))
[12, -4, 16, 8, -10]
```

- filter ([function], [iterable])
  - 각 요소에 function 함수를 적용하여 참이 나오는 것만 반환

```
>>> seq = [6, -2, 8, 4, -5]

>>> list(filter(lambda x: x > 0, seq))
[6, 8, 4]
```

TODO for Today

- **과제 1: 나만의 내장 함수 & 알고리즘 함수 만들기**

- **(Optional for Idle Student)**
  - Python 알고리즘 사이트의 문제들 풀어 보기 & 예제 보기
  - 백준 (https://www.acmicpc.net/workbook/view/459)
  - Python Example

    (https://www.programiz.com/python-programming/examples)