2021 년도 2 학기 프로그래밍언어론 최종보고서

: 파이썬의 객체지향 패러다임

이화여자대학교 컴퓨터공학과

1871056 한지수

파이썬에서 언급되는 “Everything in Python is an object” 의 의미를 다음 보고서를 통해 이야기하고자 합니다. 위 개념을 파이썬의 Memory Collection 과 Semantics 내용을 포함하여 이야기하고, 이러한 개념이 객체 지향 프로그래밍 (OOP)으로 확장되어 어떻게 활용되는지 설명해보고자 합니다.

Python Variables Are Pointers

파이썬은 등호(=) 기호를 통해 변수를 할당합니다.

```
# 변수 x 에 4 를 할당
x = 4
```

많은 프로그래밍 언어에서 변수를 데이터를 담는 컨테이너(container)나 버킷(bucket)으로 생각합니다. C 언어를 예를 들어 이야기를 해보겠습니다.

```
// C 언어 예시
int x = 4;
```

C 언어에서는 x 라는 “메모리 컨테이너(Memory container)” 를 정의합니다. 이 메모리 컨테이너는 정수 4 를 가리키는 메모리 주소를 저장합니다. C 언어와 다르게 파이썬은 변수들을 메모리 컨테이너로 생각하지 않고 포인터(Pointer)로 생각합니다. 다음 코드를 보면,


```
x = 4
```

해당 코드는 4 의 값을 가지는 컨테이너를 가리키는 포인터 x 를 정의하고 있습니다. 파이썬 변수는 다양한 객체를 가리키기 때문에 (C 언어처럼) 변수를 선언할 필요가 없으며 변수가 항상 같은 유형의 정보를 가리키도록 요구할 필요도 없습니다. 변수 이름은 모든 유형의 객체를 가리킬 수 있기에 파이썬은 dynamically-typing(동적 타이핑) 한 속성을 가지고 있음을 알 수 있습니다. 다음과 같이 파이썬이 dynamically-typed 하게 작동됩니다.


```
x = 1 # x 는 정수입니다.
x = ‘hello’ # 이제 x 는 문자열입니다.
x = [1,2,3] # 이제 x 는 목록입니다.
```

이러한 동적 타이핑(Dynamic typing) 속성으로 Python 을 매우 빠르게 작성하고 읽기 쉽게 만듭니다.

Everything in Python is an object

파이썬은 순수 객체 지향 프로그래밍 언어이며 모든 것이 객체(object)로 작동됩니다. 앞서 파이썬의 변수는 단순히 포인터라고 이야기하였습니다. 이는 변수 이름 자체에는 첨부된 type 정보가 없다고 생각할 수 있습니다. 이때문에 파이썬을 type-free 언어라고 생각할 수 있지만 사실이 아닙니다. 다음 예시를 보도록 하겠습니다:

```
>>>> print(type(2))
<class 'int'>
>>>> print(type(2.3))
<class 'float'>
>>>> print(type(2+3j))
<class 'complex'>
>>>> print(type("string"))
<class 'str'>
>>>> print(type([]))
```

```
<class 'dict'>
print(type({}))
<class 'list'>
print(type(()))
<class 'tuple'>
```

파이썬에서 클래스 개념의 type 이 존재함을 볼 수 있습니다. 이 때 type 은 변수 이름이 아니라 객체 자체에 연결됩니다. 파이썬과 같은 객체 지향 프로그래밍 언어는 객체를 메타데이터(Meta data)와 기능 (Functionality)를 포함한 entity 로 정의합니다. 파이썬에서는 모든 것이 객체입니다. 즉, 모든 객체는 그의 메타데이터 (attribute)과 관련 기능 (method)를 가지고 있습니다. 이러한 attribute 와 method 는 점(“.”) 문법을 통해 접근할 수 있습니다.

```
>>>> L = [1, 2, 3]
>>>> L.append(100)
>>>> print(L)
[1, 2, 3, 100]
```

이처럼 파이썬에서는 모든 것이 객체로 작동됩니다. 기본 자료형, 클래스, 함수 순으로 객체의 형태를 살펴보겠습니다.

1. 기본 자료형

List 와 같이 복잡한 객체에 대해서만 속성과 메서드가 있을 것으로 생각할 수 있겠지만 파이썬은 간단한 유형에도 속성과 메서드가 속성과 메서드가 연결되어 있습니다. 가장 기본적인 정수형 객체의 type 를 다음 예시로 살펴보겠습니다.

```
>>>> import sys
>>>> print(sys.maxsize)
9223372036854775807
>>>> print(type(sys.maxsize))
<class 'int'>
>>>> print(type(sys.maxsize+1))
<class 'int'>
```

다음과 같이 정수형 자료형 객체도 클래스 ‘int’라는 클래스에 속해 있음을 알 수 있습니다. (참고: 파이썬 3 은 Int 자료형의 오버플로우를 예방하여 long 이 int 에 포함됩니다. 파이썬 3 에서는 int 범위를 넘어가는 정수도 type Int 로 취급됩니다.) ‘int’ 클래스에 관해 어떻게 서술하는지 보겠습니다.

```
>>>> help(int)
Help on class int in module builtins:
class int(object)
 |  If x is not a number or if base is given, then x must be a string,
 |  bytes, or bytearray instance representing an integer literal in the
 |  given base.  The literal can be preceded by '+' or '-' and be surrounded
 |  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
 |  Base 0 means to interpret the base from the string as an integer literal.
 |  >>> int('0b100', base=0)
 |  4
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |  ##########생략##########
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
```

```
 |  from_bytes(bytes, byteorder, *, signed=False) from builtins.type
 |      Return the integer represented by the given array of bytes.
 |  ##########생략##########
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |  ##########생략##########
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  denominator
 |      the denominator of a rational number in lowest terms
 |  ##########생략##########
```

다음과 같이 가장 기본적인 자료형인 int 정수형도 클래스의 기본적인 서술과 더불어 attributes, methods, data descriptor 모두 정의되어 있음을 알 수 있습니다. 이 덕분에 정수형 자료형 객체는 (새로운 함수나 정보를 생성할 필요 없이) 내장된 attribute 와 method 을 활용하여 여러가지 작업을 수행할 수 있습니다.

```
>>>> x = 4.5
>>>> print(x.real, "+", x.imag, 'i') #class attribute
4.5 + 0.0 i
>>>> x = 4.5
>>>> x.is_integer() #class method
False
>>>> type(x.is_integer())
builtin_function_or_method
```

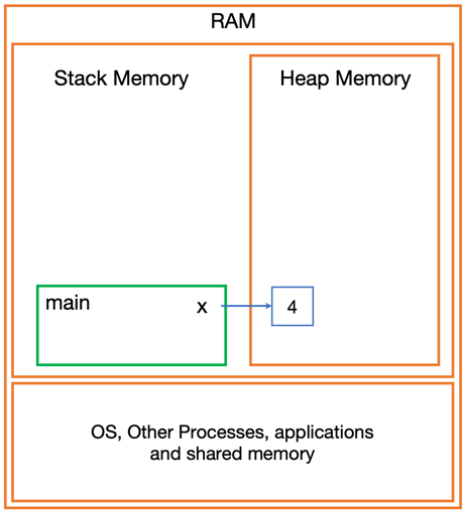
이처럼 파이썬은 복잡한 자료형부터 시작하여 간단한 자료형까지 모든 것이 객체로 이루어져 있음을 알 수 있습니다. 심지어 객체의 속성과 메서드도 자체 type 정보를 가진 객체임을 볼 수 있습니다.

* 기본 자료형의 메모리 할당(Memory allocation)

기본 자료형의 메모리 할당이 어떻게 되는지 이야기하기 전에 파이썬의 메모리 구조에 대해 이야기해보겠습니다.

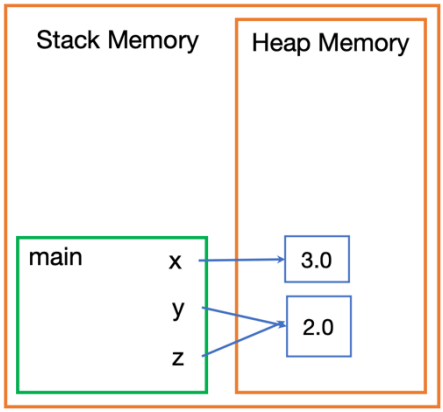
오른쪽 그림은 RAM 상의 메모리 구조를 나타낸 그림입니다. RAM 상에서 크게 두 가지 메모리로 나뉩니다. OS 와 기타 앱들, 공유메모리와 이를 제외한 메모리로 나뉩니다. 이 때 맨 밑의 메모리를 제외한 위의 Stack Memory 와 Heap Memory 에서 메모리 할당이 이루어집니다. x=4 라는 간단한 코드를 실행하였을 때 오른쪽 그림과 같이 Stack Memory 의 main 공간 위에 변수 x(: 포인터)가 생성되고 이 변수는 Heap Memory 의 해당 객체 와 연결되는 구조를 가지고 있습니다.

즉, Stack Memory 는 함수가 실행될 때 영역이 생성되며 필요한 변수들이 올라오게 되며, Heap Memory 는 필요한 객체들이 올라오며 포인터로 연결됨을 알 수 있습니다. (다음 메모리 할당 관련 설명부터는 중요하게 작용되는 Stack Memory 와 Heap Memory 만을 나타냅니다.)



다음 응용 코드와 함께 메모리상에서의 변수 상황을 보겠습니다.

```
>>>> x = 2.0
>>>> y = x
>>>> if (id(x) == id(y)):
>>>>     print("x and y refer to the same object")
x and y refer to the same object
>>>> x = x + 1
>>>> if (id(x) == id(y)):
>>>>     print("x and y refer to the DIFFERENT object")
x and y refer to the DIFFERENT object
>>>> z = 2.0
>>>> if (id(y) == id(z)):
>>>>     print("y and z refer to the SAME object")
y and z refer to the SAME object
```



코드 상에서 y 와 z 가 float 형 객체 2.0 을 각각 가리키고 있습니다. 파이썬에서는 이 둘이 동일한 객체를 나타내므로 포인터 값 또한 동일하게 작용됨을 알 수 있습니다. (파이썬에서 id 라는 함수는 객체의 고유값(레퍼런스)를 나타내는 개념입니다.) 이렇게 **간단한 자료형도 객체로 이루어져 메모리 관리가 용이하게 이루어짐**을 알 수 있습니다.

2. 함수: Functions are first-class object

일반적인 함수를 통해 함수는 어떤 형태로 정의되는지 살펴보도록 하겠습니다.

```
def area(h, v):
    return h*v
print(type(area))
>>>> <class 'function'>
```

일반적인 함수를 이야기하면 함수가 클래스 'function' 에 정의되어 있음을 알 수 있습니다. 파이썬은 함수를 정의할 때 사용자가 정의한 함수 객체를 정의합니다. 이를 "Functions are first-class object" 라 이야기합니다. 다음 예시들을 통해 함수 객체를 어떻게 사용할 수 있는지 보겠습니다.

```
>>>> def yell(text):
>>>>     return text.upper() + '!'
>>>> yell('hello')
'HELLO!'
>>>> bark = yell #함수를 다른 변수에 할당
>>>> bark('woof)
'WOOF!'
>>>> del yell # 함수 객체와 변수명은 다른 개념
>>>> yell('hello?')
NameError: name 'yell' is not defined
>>>> bark('hey')
'HEY!'
```

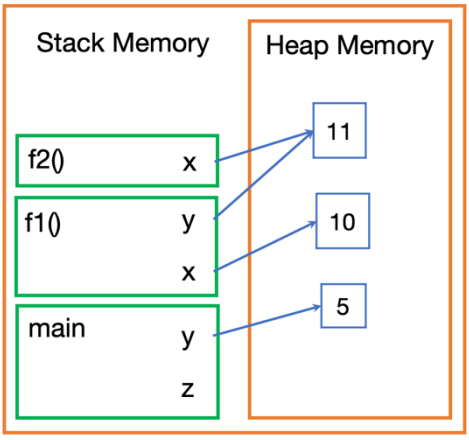
함수 yell 을 정의하였다면 이는 객체 yell 이 생성된 것입니다. yell 함수가 객체이기 때문에 yell 을 다른 변수에 할당할 수 있습니다. bark=yell 은 함수를 호출하지 않고 yell 과 연결되어있는 함수 객체를 가져와 그 객체를 가리키는 새로운 변수인 bark 를 만들었다고 할 수 있습니다. 이 덕분에 같은 함수를 bark 라는 객체로 호출이 가능합니다.

함수 객체와 변수명이 아예 다른 개념임을 del yell 코드부터 알 수 있습니다. 같은 함수를 가리키는 다른 이름인 bark 라는 객체를 가지고 있기에 원래 이름인 yell 을 지워도 아무 문제 없이 bark 가 잘 실행됨을 알 수 있습니다.

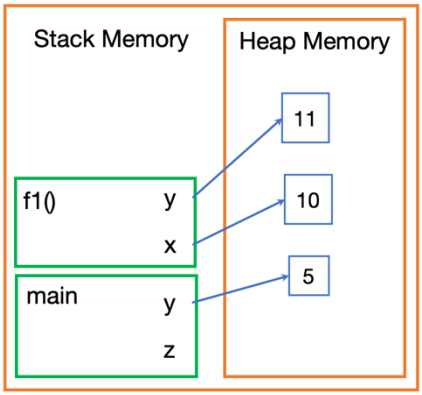
* 함수의 메모리 할당

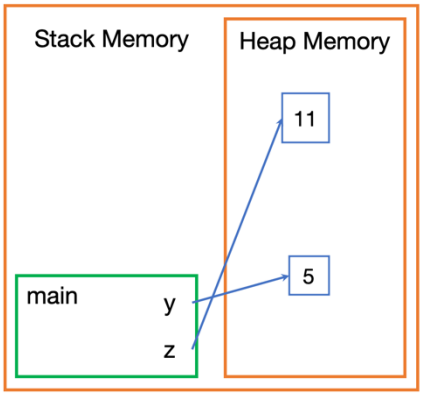
위의 예제들을 통해 파이썬의 함수는 객체이고, 이에 따른 응용 코드들을 보았습니다. 함수의 메모리 할당은 어떻게 일어나는지 살펴보도록 하겠습니다.

```
# f2
def f2(x):
        x = x + 1
        return x
#f1
def f1(x):
        x = x * 2
        y = f2(x)
        return y
#main
y = 5
z = f1(y)
```



함수를 호출하게 되면 stack 영역에 영역이 생성되며 변수들은 포인터로 작동됩니다. 오른쪽 그림은 호출된 f2 함수가 실행되는 동안 memory 의 작업 상황입니다. f1, f2 함수의 stack 영역이 생성되고 f2 함수가 먼저 실행되어 값에 해당되는 객체들을 가리키는 것을 볼 수 있습니다. **기본적인 자료형을 가지는 객체들로 연결되어있다 보니 연결과 할당이 용이하여 결국 함수도 관리가 용이함**을 볼 수 있습니다. 메모리 할당 해제 과정도 같이 보도록 하겠습니다.





왼쪽 그림은 f2 함수가 해제되고 f1 함수의 실행되는 동안의 작업 상황, 오른쪽 그림은 최종 배정 결과입니다. f1, f2 함수가 순서대로 실행이 끝남에 따라 stack memory 에서 사라지며 아무도 가리키지 않는 객체 10 도 사라지는 상황을 볼 수 있습니다. **포인터를 통한 객체 관리가 용이하다 보니 함수를 통한 Garbage collection 에서도 이점을 보이고, 최종적으로 효율적인 메모리 관리를 이끌어낸다는 장점**이 있습니다.

3. 클래스

자료형, 함수에 이어 마지막으로 클래스를 구현 어떻게 객체로 작동되는지 알아보겠습니다. 클래스 정의는 앞선 예비 기말프로젝트에서 정의한 클래스를 사용하겠습니다.

```
class Dog:
    #Class Variable
    species = "Canis familiaris"

    def __init__(self, name, age):
```

```
        #Instance Variable
        self.name = name
        self.age = age
        self.leg = 4

    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old"

    # Another instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"
```

추가적으로 예비 프로젝트에 클래스 정보를 자세히 설명하였으므로 생략하고 곧바로 객체 정의를 하겠습니다.

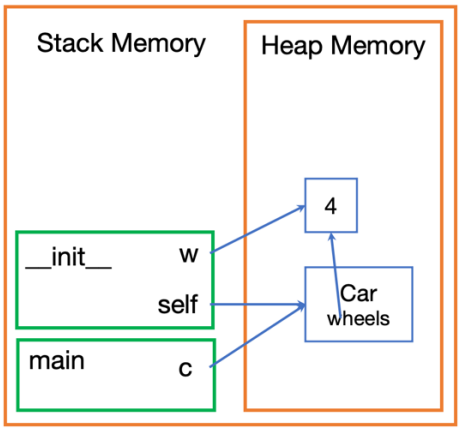
```
buddy = Dog("Buddy", 9)
print(type(buddy))
>>>> <class '__main__.Dog'>
print(type(buddy.name))
>>>> <class 'str'>
print(type(buddy.description()))
>>>> <class 'str'>
```

다음과 같이 현재 실행되고 있는 파일에 Dog 라는 클래스를 바탕으로 객체가 생성되며, 객체의 attribute 와 method 를 활용할 때 모두 객체로 실행됨을 알 수 있습니다.

* 클래스의 메모리 할당

클래스는 메모리 할당이 어떻게 되는지 다음 예제를 통해 알아보도록 하겠습니다.

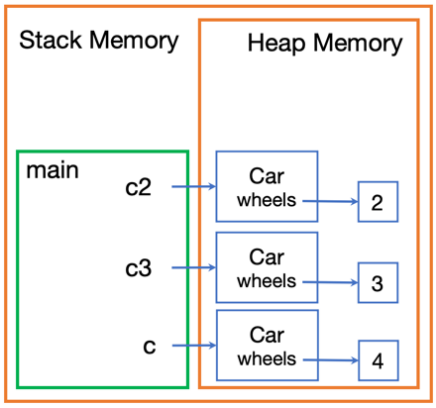
```
class Car:
    def __init__(self, w):
        self.wheels = w

    def getWheels(self):
        return self.wheels

# main
c = Car(4)
```



간단한 Car 클래스 정의와 객체를 생성한 예제입니다. 오른쪽은 클래스 객체를 생성하였을 때 메모리 상황입니다. main 함수에서 c=Car(4) 인 클래스 객체를 생성하기에 생성자 함수 __init__이 호출됩니다. 함수 객체가 생성되므로 Stack memory 영역에 생성됩니다. w 는 4 이고, self 은 Car 인스턴스의 주소값으로 작용합니다. (self 는 항상 자신의 클래스 인스턴스 주소 값으로 작용합니다.) self.wheels 은 객체의 attribute 에 속하므로 Heap memory 에 할당되어 존재합니다. 객체 여러 개를 생성해보겠습니다.

```
c = Car(4)
c3 = Car(3)
c2 = Car(2)
```

앞선 코드를 실행해보면 오른쪽 그림과 같이 메모리가 구성됩니다. Heap Memory에서 객체가 생성되고 이에 따른 매개변수들은 객체들로, 이를 따로 생성하고 포인터로 연결하는 방식으로 이루어집니다.



이처럼 파이썬은 기본적인 자료형부터 시작하여 함수, 클래스 모두 객체로 이루어져 있습니다. 파이썬의 객체 지향 프로그래밍(Object-Oriented Programming) 패러다임은 사물 간의 관계를 모델링하기, 메모리 관리도 용이하며 쉬운 프로그램의 개발로 생산성을 향상시킬 수 있다는 장점을 가지고 있습니다.

객체지향프로그래밍이란?

앞서 파이썬의 모든 것이 객체라고 정의한 바 있습니다. 우리가 일반적으로 사용했던 숫자나 리스트, 튜플 등 모든 자료형이 어떤 클래스의 인스턴스입니다. 코드로 본 바와 같이 인스턴스들은 특정 클래스에 속해있고 그 클래스들은 특정 속성(Attribute)와 기능(Method)를 가짐을 알 수 있습니다. 클래스들의 속성과 기능을 미리 정의한 부분을 통해 볼 수 있듯이 객체 지향프로그래밍은 클래스를 구성하기 위해 관련된 속성이나 동작을 개별 객체로 묶음으로써 구조화된 프로그램을 구성하는 방법입니다. 예를 들어, 객체는 이름, 나이, 주소와 같은 속성(Attribute)와 걷기,말하기, 호흡하기, 달리기와 같은 행동(Method)을 가진 사람을 나타낼 수 있습니다. 즉, 객체 지향 프로그래밍은 자동차와 같은 구체적이고 실제적인 사물 뿐만 아니라 회사와 직원, 학생과 교사 등과 같은 사물 간의 관계를 모델링하는 접근 방식입니다.

이렇게 클래스로 구성하여 객체를 불러내는 형식으로 이용되어 문제를 쉽고 자연스럽게 프로그램화(모델링)할 수 있다는 장점을 가집니다. 이 때 객체들이 완전한 독립성이 유지되는 방식으로 진행되며, 프로그램 모듈이 모두 재사용 가능합니다. 이덕분에 (전 방식에 속하는 절차 지향 패러다임에 비해) 매우 효율적이며 직관적인 코드 구성이 가능해집니다.

* 객체지향프로그래밍의 기능성 확장하기

OOP 패러다임의 가장 큰 특징 중 하나는 함수를 통한 기능성 확장이 가능하다는 점입니다. 기능성 확장으로 불필요한 코드 작업을 줄여 효율적인 코드 작성, 효율적인 메모리 관리가 가능합니다. 대표적인 특징인 상속 (inheritance)와 다형성 (Polymorphism)을 예시들을 통해 확인해보겠습니다.

1. 상속(Inheritance)

상속은 이미 부모 클래스 용으로 작성된 클래스를 바탕으로 자식 클래스 코드를 작성하는 방법입니다. 예를 들어, 부모 클래스인 Car 클래스의 일부 속성은 자식 클래스인 Bus, Truck 등의 클래스와 동일합니다. 상속 특성을 통해 자식 클래스는 부모 클래스의 기능을 확장할 수 있다는 장점이 있습니다. 즉, 자식 클래스는 부모의 모든 속성과 메서드를 상속하지만 자신에게 고유한 속성과 메서드를 지정할 수 있습니다.

2. 다형성(Polymorphism)

자식 클래스와 부모 클래스의 inheritance 관계를 통해 자식 클래스는 메서드를 재정의할 수 있습니다. 다형성은 다양한 형태를 취할 수 있는 능력을 의미합니다. 이러한 다형성은 메서드 오버라이딩 (Method Overriding)과 메서드 오버로딩(Method Overloading) 으로 존재합니다. 위에서 서술된 개 클래스를 바탕으로, 오버라이딩된 다음 예제를 확인해보겠습니다.

```
class GoldenRetriever(Dog):
    def speak(self, sound="Bark"):
        return super().speak(sound)
```

개의 품종 중 하나인 GoldenRetriever 에 대한 자식 클래스입니다. 개 품종마다 짖는 소리가 약간 다르기 때문에 부모 클래스에 나온 .speak() 함수를 재정의해야 합니다. GoldenRetriever 클래스에서는 부모 클래스인 Dog 의 메서드를 상속하여 그대로 사용 가능합니다. 자식 클래스에 대한 객체를 생성해보고 함수를 실행한 결과를 보겠습니다.

```
>>> jim = GoldenRetriever("Jim", 5)
>>> jim.speak()
'Jim says Bark'
```

코드 실행 시 Jim 라는 5 살 된 골든 리트리버 품종의 개 인스턴스가 생성됩니다. 메서드 오버라이딩을 통해, 이전에 Dog 객체에서 speak 함수를 부를 때 sound 파라미터를 넣을 필요없이 품종에 따라 바로 함수가 작동됨을 알 수 있습니다.

객체지향패러다임의 대표적인 특징들을 예시와 함께 살펴보았습니다. 상속을 통해 기능성이 확장되면서 응집력이 강화하여 효율적이며 직관적인 코드를 가능케 합니다. 다형성으로 클래스 간에 독립적으로 디자인하여 결합력을 약하게 할 수 있다는 면이 존재합니다.