Python and Libraries

2025.8

(*) Reference

- Wes Mckinney, Python for Data Analysis: Data Wrangling with Pandas, NumPy, and Ipython, O'Reilly, 2012
- Many Internet sites, ChatGPT, Gemini

Why Python for Data Science?

- 직관적인 문법과 높은 생산성
  - 마치 영어처럼 읽히는 코드로, 비전공자도 쉽게 배울 수 있다.
  - 간결한 코드로 복잡한 데이터 분석 로직을 구현할 수 있다.
- 강력하고 방대한 생태계
  - NumPy (수치 계산), Pandas (데이터 조작), Matplotlib (시각화), Scikit-learn (머신러닝) 등 데이터 과학의 모든 과정을 지원하는 수많은 라이브러리가 존재한다.
- 오픈 소스와 거대한 커뮤니티
  - 무료로 사용할 수 있으며, 전 세계 개발자들이 함께 발전시키고 있다.
  - 문제가 발생했을 때 해결책을 찾기 쉽고, 최신 기술이 가장 먼저 공유된다.

The Anaconda Ecosystem

- What is Anaconda?
  - 데이터 과학 및 머신러닝을 위한 파이썬 개발 플랫폼이다.
  - 파이썬, Jupyter Notebook, 그리고 150개 이상의 필수 라이브러리(NumPy, Pandas 등)를 한 번에 설치해주는 패키지이다.
- Why Anaconda?
  - > 400 packages available, 150 automatically installed
  - Free, open source
  - Support all major platforms
  - Very reliable and easy to use
  - Scale up to professional and commercial use (with fee)
- System requirements
  - Minimum 3 GB disk space

Jupyter Notebook

- Convenient **web-based executable script files**
  - Interactive code development
  - Cell-wise execution
  - No reloading of script (.py) files necessary
  - Easy to share
  - Excellent teaching tool
- Project Jupyter was born out of the IPython Project in 2014
  - Jupyter can support (or be interfaced with) other languages (Ruby, R, Julia, etc.)
- Requires Google Chrome or Mozilla Firefox
- On-line examples
  - https://nbviewer.jupyter.org

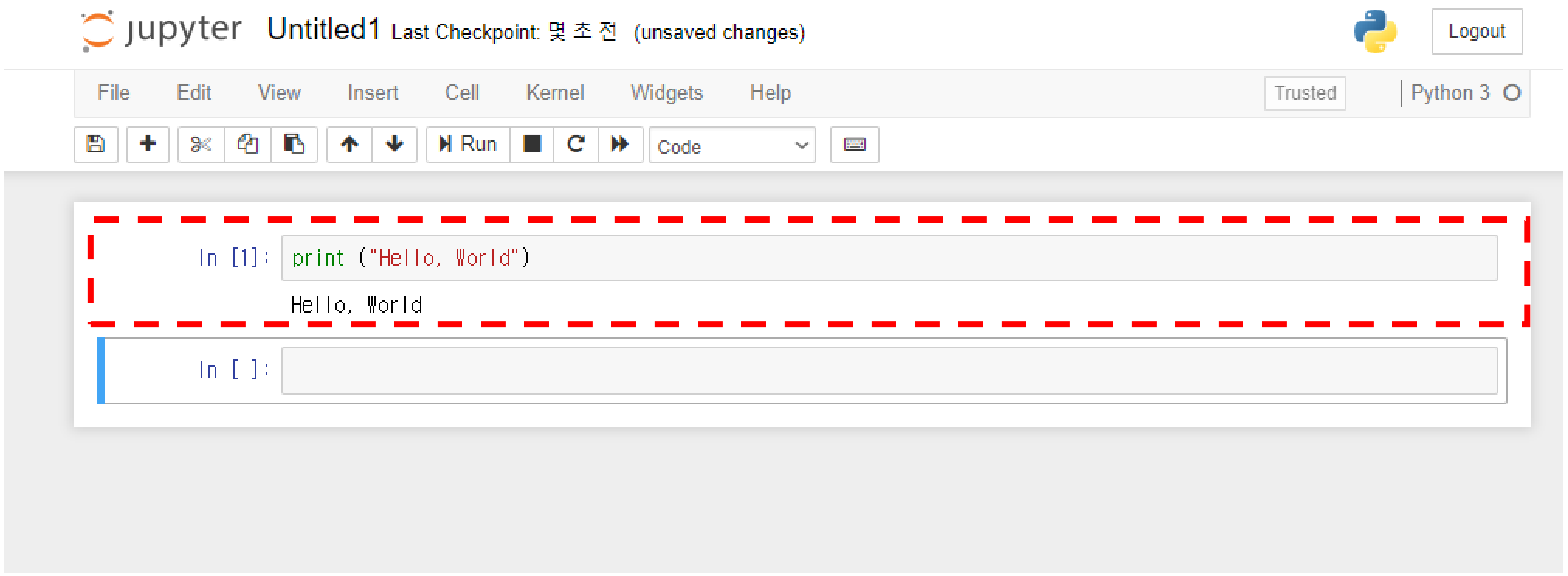
JupyterLab

- JupyterLab 이란?
  - Jupyter Notebook의 차세대 버전으로, 더 통합된 인터페이스를 제공하는 웹 기반 대화형 개발 환경.
- 데이터 분석에 강력한 환경
  - 셀(Cell) 단위 실행: 코드 조각을 독립적으로 실행하고 즉시 결과를 확인할 수 있어, 데이터를 탐색하고 실험하는 데 이상적.
  - 시각화: 코드 실행 결과로 생성된 표나 그래프를 바로 아래에서 확인할 수 있다.
  - 문서화: 코드와 함께 설명, 수식, 이미지를 마크다운(Markdown)으로 작성하여, 분석 과정을 하나의 완성된 문서처럼 만들고 공유하기 편리하다.
- 주요구성 요소
  - 노트북, 콘솔, 터미널, 텍스트 편집기

Jupyter Notebook or Lab

- To start, from command line, enter “jupyter notebook” or click the icon “Jupyter Notebook” from startup menu and set the type as “Python 3”.
- For JupyterLab, enter “jupyter lab”. (You can go to your working directory before entering JupyterLab, or you can navigate to your working directory in JupyterLab.)
- You will have either “Coding” cell or “Markdown” cell. (Jupyter Notebook)
- You can access “Notebook”, “Console”, “Text File”, “Markdown”, “Terminal”. (JupyterLab)
- Markdown (documentation) guides:
  - https://colab.research.google.com/notebooks/markdown_guide.ipynb

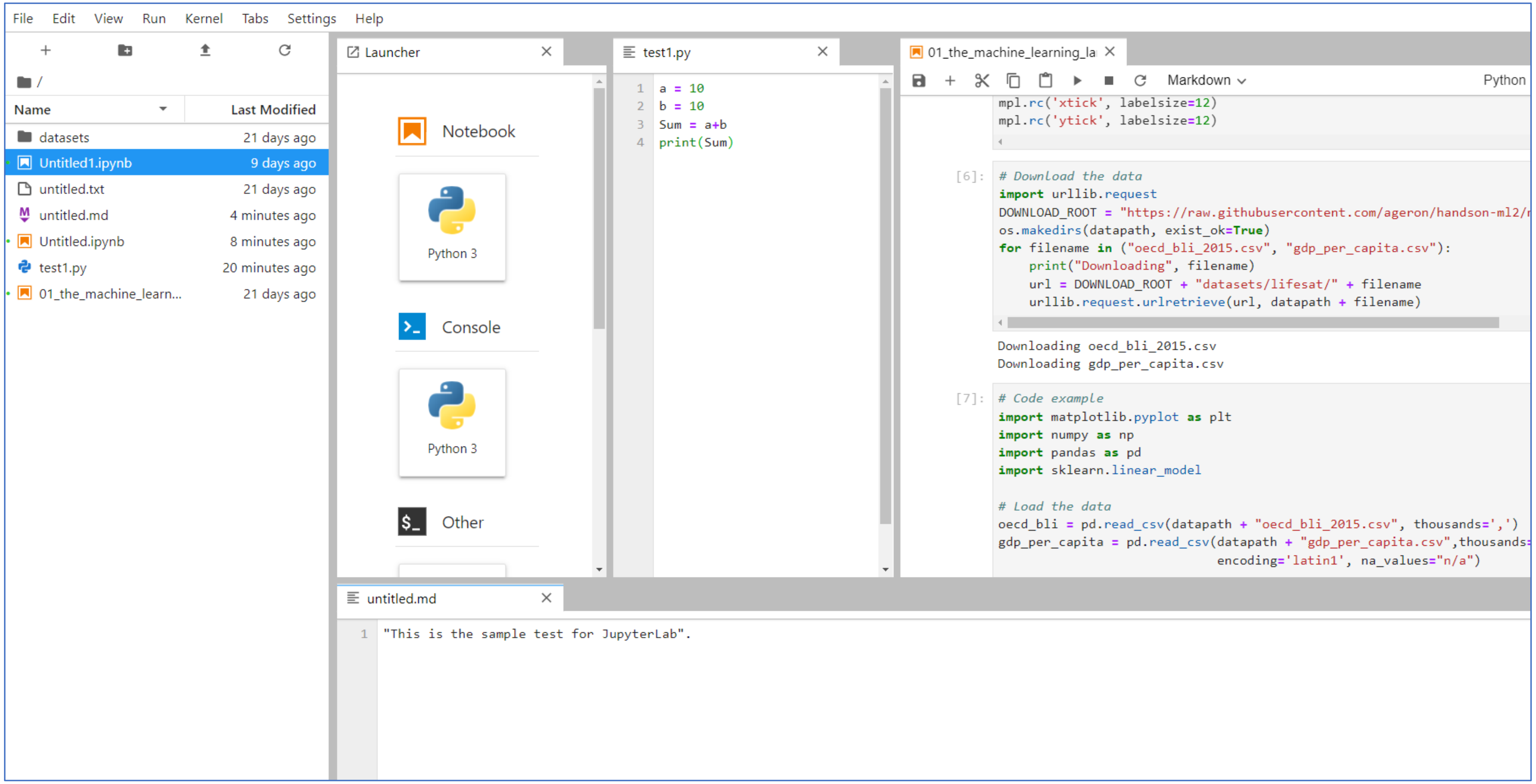
Jupyter notebook Python 3



```
In [1]: print ("Hello, World")
        Hello, World
In [ ]:
```

❖ In place: Ctrl + Enter
❖ To execute cell and move to next cell: Shift + Enter
  • Create new cell if necessary
❖ To execute and insert new cell: Alt + Enter

JupyterLab



Name	Last Modified
datasets	21 days ago
Untitled1.ipynb	9 days ago
untitled.txt	21 days ago
untitled.md	4 minutes ago
Untitled.ipynb	8 minutes ago
test1.py	20 minutes ago
01_the_machine_learn...	21 days ago

```
a = 10
b = 10
Sum = a+b
print(Sum)
```

```
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)
```

```
[6]: # Download the data
import urllib.request
DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/r
os.makedirs(datapath, exist_ok=True)
for filename in ("oecd_bli_2015.csv", "gdp_per_capita.csv"):
    print("Downloading", filename)
    url = DOWNLOAD_ROOT + "datasets/lifesat/" + filename
    urllib.request.urlretrieve(url, datapath + filename)
```

```
Downloading oecd_bli_2015.csv
Downloading gdp_per_capita.csv
```

```
[7]: # Code example
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn.linear_model

# Load the data
oecd_bli = pd.read_csv(datapath + "oecd_bli_2015.csv", thousands=',')
gdp_per_capita = pd.read_csv(datapath + "gdp_per_capita.csv",thousands=
                             encoding='latin1', na_values="n/a")
```

```
"This is the sample test for JupyterLab".
```

JupyterLab

- What is JupyterLab?
  - a next-generation web-based user interface for Project Jupyter
  - Fully support Jupyter Notebook
  - Enables users to use text editor, terminals, data file viewers, and others
- Installation (in command shell)
  - conda install –c conda-forge jupyterlab # when using conda
  - pip install jupyterlab # when using pip
- Supported browsers
  - Firefox, Chrome, Safari
- Starting jupyterlab
  - Type “jupyter lab” in command shell

Keyboard Shortcuts - Jupyter

- Command mode (press ESC to enable)

In command mode	
Shift-Enter	run cell, select below
Ctrl-Enter	run selected cells
Alt-Enter	run cell and insert below
a/b	insert cell above/below
x/c	cut selected cells / copy selected cells
Shift-v / v	paste cells above/below
Shift-m	merge selected cells, or current cell with the cell below if only one cell is selected

In command mode	
l	toggle line numbers
o	toggle output of selected cells
h	show keyboard shortcuts
Shift-Space	scroll notebook up
Space	scroll notebook down
Window-/	toggle comment

In edit mode (press Enter)	
Ctrl-Shift-Minus	Split cell at cursor

Notebook Cell Types

- Code cells
  - Edit and execute cells inline, generates output as text, figures, HTML tables
  - Syntax highlighting, tab completion, introspection
  - Default for inserted cells
- Markdown cells
  - Rich text input, including HTML and LaTex
  - Cell replaced by text output when executed (**Documents**)
- Raw text cells
  - Executed as input (no formatting)
  - Cell remains in place
- Heading cells
  - Levels 1 through 6, similar to Microsoft Word
  - Can be used to generate Table of Contents

Colab from Google

- Free cloud service from Google
  - A Jupyter notebook environment that requires no setup to use
  - Supports free GPU/TPU, and Runs entirely in the cloud
  - provides a maximum GPU runtime of 8~12 hours ideally at a time
- Useful Shortcuts

actions	colab	jupyer
Convert to code cell	Ctl-M Y	Y
Convert to text cell	Ctl-M M	M
Split at cursor	Ctl-M – (minus sign)	Ctrl Shift -
Merge two cells	Ctl-M /	Shift M
Show keyboard shortcuts	Ctl-M H	H
Interrupt execution	Ctl-M I	II

What is Python Language?

- Completely open source, started in early 1990
- **Script language (interpreter)** , i.e. no compiler
  - Directly translate source code (do not generate compiled code)
  - Converted to (platform-independent) bytecode (and Python Virtual Machine(PVM) interprets and executes it – slow)
  - **Interpreters translate the source code to bytecode and then execute that bytecode. (Python, Java)**
- **Easy, Very portable**, mostly runnable on all supported platforms
- **Object-oriented** and Functional
- **Large standard libraries** with huge set of external modules
- **Dynamically typed:** variable type determined at **run-time** (no need of variable declaration), hence slow… but efficient memory usage

Python Scripts

- **Python script**: Collection of commands in a file designed to be executed like a program
- Use any editor to create a Python script, say, *myscript.py*
- No compilation needed
  - Python script is **interpreted**. More precisely, it is converted to byte code (.pyc), and then executed.
- Two running modes:
  - Running the Python script (from the command line)
    - % python myscript.py
    - (ex) calculator, running scripts, test environment
  - Running the Python interactively in a shell (e.g., a Jupyter Notebook Notebook or IDE)
    - **Jupyter** or Spyder, or other IDE
    - (ex) work processes (ideal for data processing and analysis), documentation, teaching or presentation

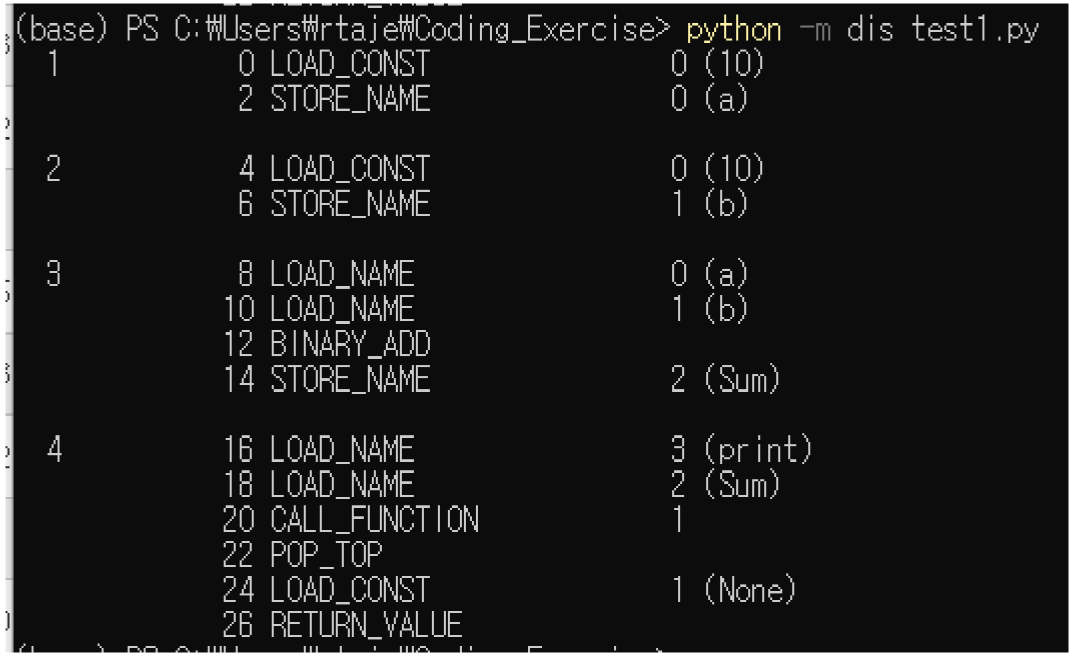
How Python Works: Interpreter

- 파이썬은 인터프리터(Interpreter) 언어이다.
  - C언어처럼 코드 전체를 기계어로 번역(컴파일)한 후 실행하는 방식이 아니다.
  - 인터프리터가 우리가 작성한 코드를 한 줄씩 읽어 바이트코드(Bytecode) 라는 중간 언어로 변환하고, 파이썬 가상 머신(PVM)이 이 바이트코드를 즉시 실행한다.
- 동시통역사 vs. 번역가
  - 인터프리터 (파이썬): 말이 나올 때마다 바로 통역하는 '동시통역사'
  - 컴파일러 (C언어): 연설문 전체를 받아 번역본을 완성하는 '번역가'

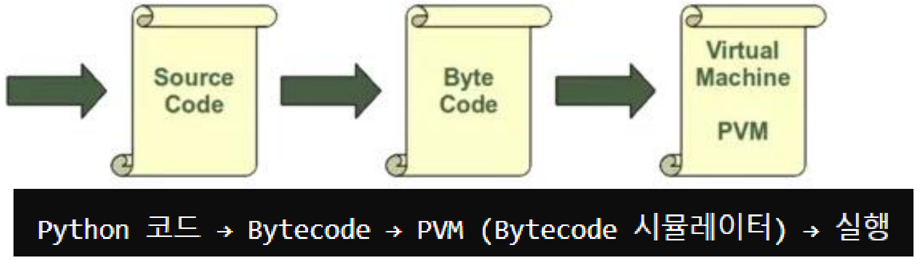
Python Execution

- Python is an **interpreted language** and executed as:
  - Step 1 : (**compilation**) reads python code (.py file) and checks the syntax, and translate it into its equivalent form in intermediate language, “Byte code”. (.pyc file)
  - Step 2: (**interpretation**) Python Virtual Machine(PVM) converts the byte code into machine-executable code.

```
a = 10
b = 10
Sum = a+b
print(Sum)
```



```
(base) PS C:₩Users₩rtaje₩Coding_Exercise> python -m dis test1.py
  1           0 LOAD_CONST               0 (10)
              2 STORE_NAME               0 (a)

  2           4 LOAD_CONST               0 (10)
              6 STORE_NAME               1 (b)

  3           8 LOAD_NAME                0 (a)
             10 LOAD_NAME                1 (b)
             12 BINARY_ADD
             14 STORE_NAME               2 (Sum)

  4          16 LOAD_NAME                3 (print)
             18 LOAD_NAME                2 (Sum)
             20 CALL_FUNCTION            1
             22 POP_TOP
             24 LOAD_CONST               1 (None)
             26 RETURN_VALUE
```

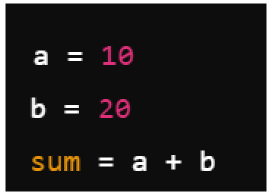


Byte code example

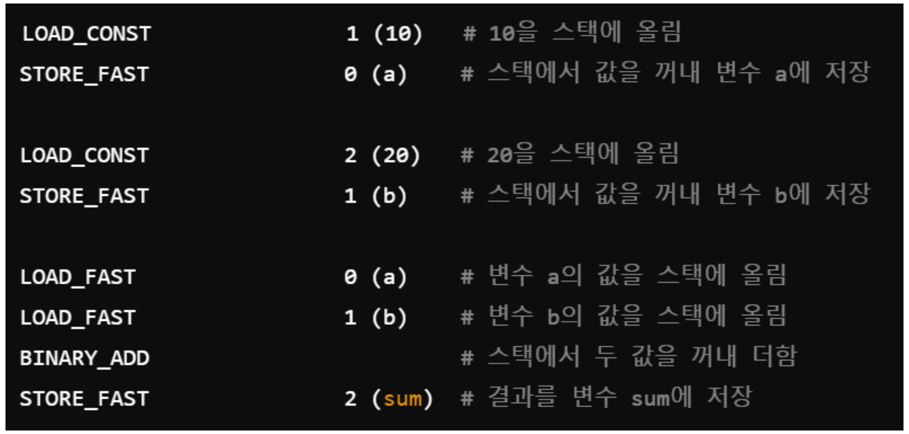
1. Line Number
2. offset position of byte code
3. name of byte code instruction
4. instruction’s argument
5. constants or names (in brackets)

Python Execution

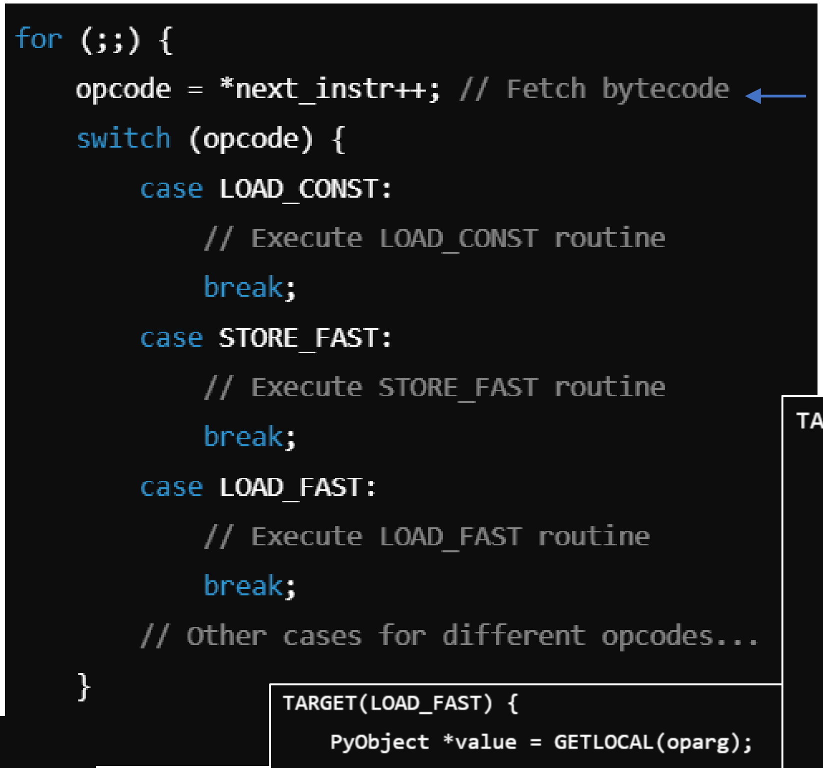
- Python code -> Bytecode -> Execution (corresponding C-routines inside the interpreter)



```python
a = 10
b = 20
sum = a + b
```

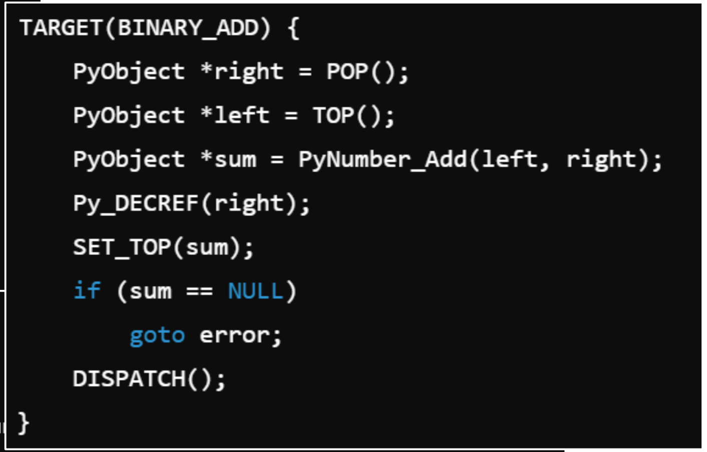


```
LOAD_CONST             1 (10)    # 10을 스택에 올림
STORE_FAST             0 (a)     # 스택에서 값을 꺼내 변수 a에 저장

LOAD_CONST             2 (20)    # 20을 스택에 올림
STORE_FAST             1 (b)     # 스택에서 값을 꺼내 변수 b에 저장

LOAD_FAST              0 (a)     # 변수 a의 값을 스택에 올림
LOAD_FAST              1 (b)     # 변수 b의 값을 스택에 올림
BINARY_ADD                       # 스택에서 두 값을 꺼내 더함
STORE_FAST             2 (sum)   # 결과를 변수 sum에 저장
```

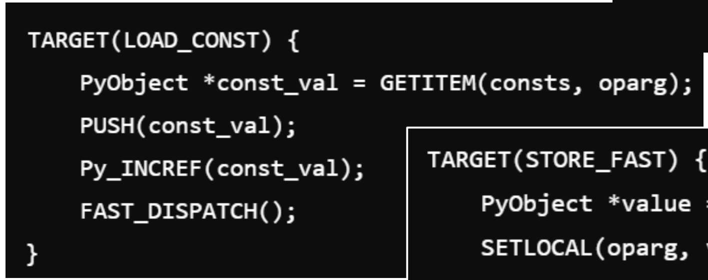


```c
for (;;) {
    opcode = *next_instr++; // Fetch bytecode
    switch (opcode) {
        case LOAD_CONST:
            // Execute LOAD_CONST routine
            break;
        case STORE_FAST:
            // Execute STORE_FAST routine
            break;
        case LOAD_FAST:
            // Execute LOAD_FAST routine
            break;
        // Other cases for different opcodes...
    }
```

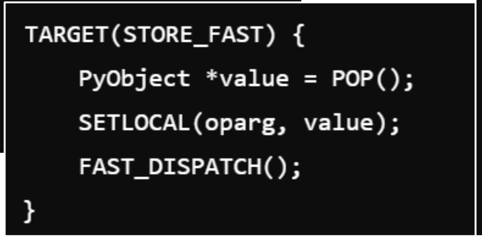
evaluation loop inside the interpreter (found in ceval.c)



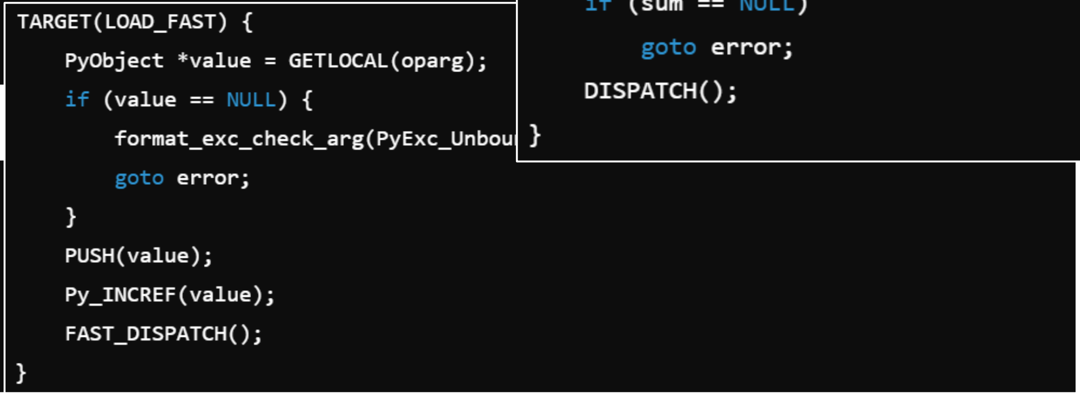
```c
TARGET(BINARY_ADD) {
    PyObject *right = POP();
    PyObject *left = TOP();
    PyObject *sum = PyNumber_Add(left, right);
    Py_DECREF(right);
    SET_TOP(sum);
    if (sum == NULL)
        goto error;
    DISPATCH();
}
```



```c
TARGET(LOAD_CONST) {
    PyObject *const_val = GETITEM(consts, oparg);
    PUSH(const_val);
    Py_INCREF(const_val);
    FAST_DISPATCH();
}
```



```c
TARGET(STORE_FAST) {
    PyObject *value = POP();
    SETLOCAL(oparg, value);
    FAST_DISPATCH();
}
```

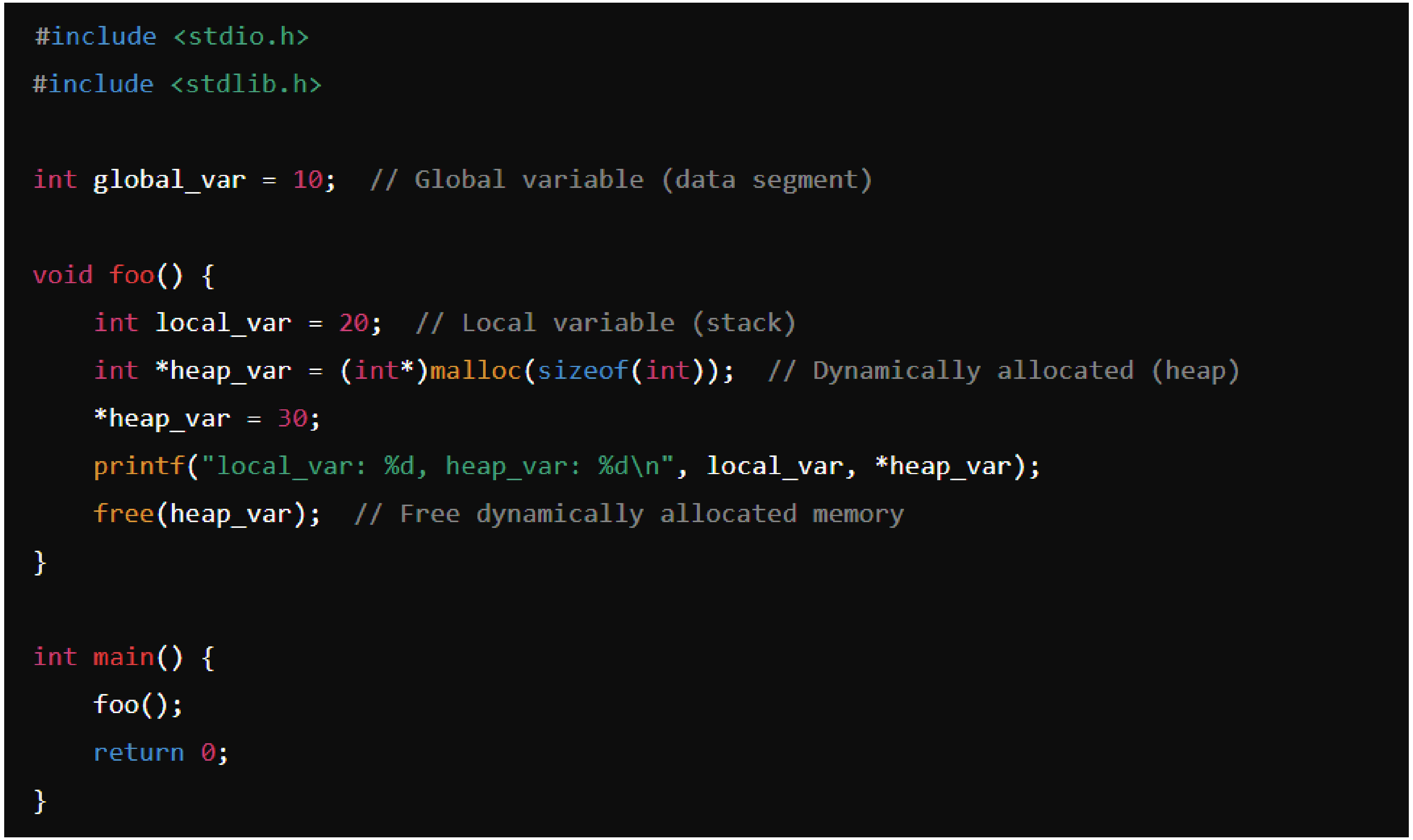


```c
TARGET(LOAD_FAST) {
    PyObject *value = GETLOCAL(oparg);
    if (value == NULL) {
        format_exc_check_arg(PyExc_Unbou
        goto error;
    }
    PUSH(value);
    Py_INCREF(value);
    FAST_DISPATCH();
}
```

C and Python

- In C:
  - Assigning variables and objects to memory is handled by a combination of the compiler and the linker.
    - **Compiler**: Translates the source code into machine code and generates information about variables and objects (such as their size, types, and locations.
    - **Linker**: Combines object files generated by the compiler and assigns them actual memory locations, resolving external references.
    - **Loader**: Loads the program into memory and starts its execution.
  - How memory is managed in C
    - Stack memory (local variables)
    - Heap memory (dynamic variables)
    - Data segment (global/static variables)
    - Code segment (program codes)
  - Memory allocation
    - Compile-time (static allocation) – global/static
    - Run-time (dynamic allocation) – malloc()

C and Python



```c
#include <stdio.h>
#include <stdlib.h>

int global_var = 10;  // Global variable (data segment)

void foo() {
    int local_var = 20;  // Local variable (stack)
    int *heap_var = (int*)malloc(sizeof(int));  // Dynamically allocated (heap)
    *heap_var = 30;
    printf("local_var: %d, heap_var: %d\n", local_var, *heap_var);
    free(heap_var);  // Free dynamically allocated memory
}

int main() {
    foo();
    return 0;
}
```

C and Python

- In Python:
  - It uses dynamic memory allocation, managed by the Python interpreter and its internal memory manager at runtime.
    - **Python Interpreter**: dynamically creates an object and assigns memory to objects (in the heap) and variables (in the stack).
    - **Memory Manager**: Python's memory manager (implemented in C) allocates memory for Python objects and releases it when it is no longer needed (through garbage collection).
    - **Garbage Collector**: to reclaim memory that is no longer being used. This is done using <u>reference counting</u> and <u>cyclic garbage collection</u> to free memory for objects that are no longer accessible.
  - How memory is managed in Python:
    - Heap memory: All objects (including integers, lists, and instances of classes)
    - Stack memory (for references): Variable references (local variables inside functions)
  - Memory allocation Process
    1. Variable assignments
    2. Object creation
    3. Garbage collection

Python Interpreter

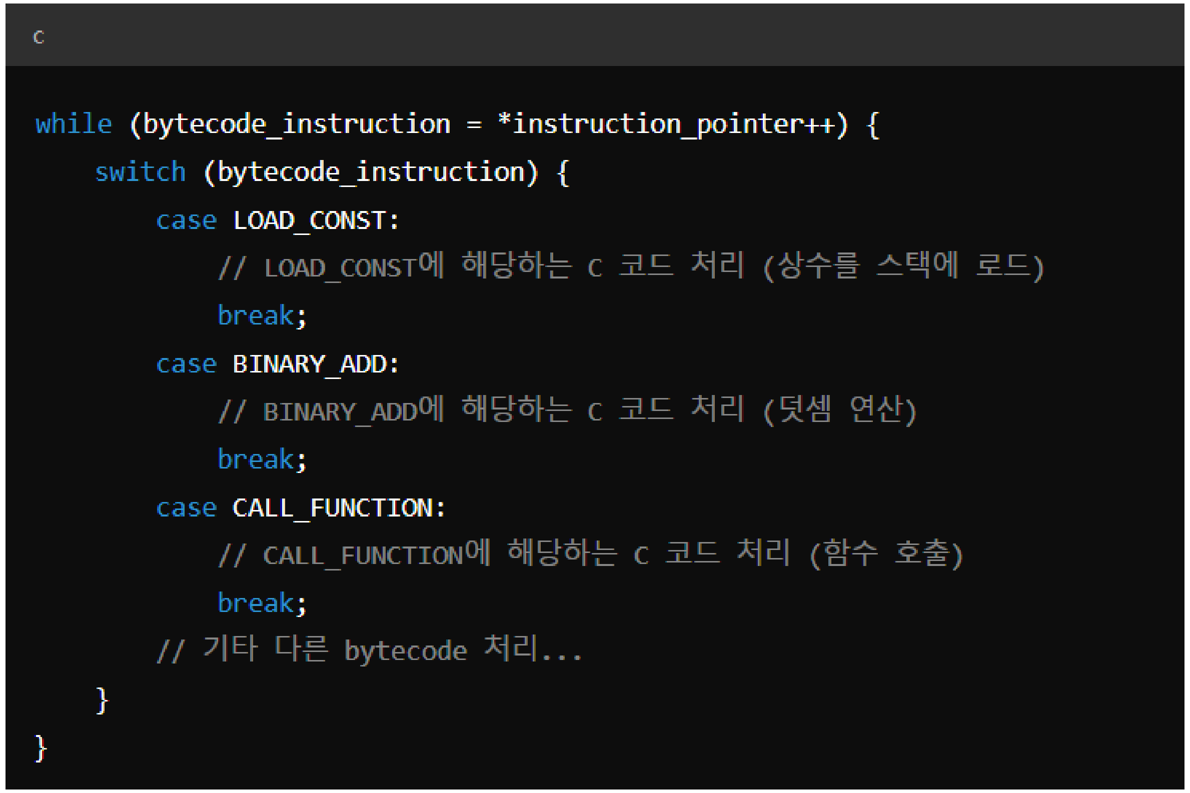
- Python Interpreter:
  - It first converts the source code into bytecode.
  - The bytecode is then executed by the Python Virtual Machine, which uses C routines to simulate the execution of the bytecode instructions. (can be thought as Bytecode Simulator)
- key components of the Python interpreter
  1. **Lexical Parser**: breaks down the source code into tokens, analyzes the token stream and determine grammatical code structure, and builds an abstract syntax tree (AST)
  2. **Compiler**: converts the AST into bytecode
  3. **Bytecode Evaluator** (Python Virtual Machine): interprets and executes the bytecode
     - It simulates the execution of bytecode.
     - And, it also handles memory management, garbage collection, and other essential tasks

Python interpreter and Bytecode

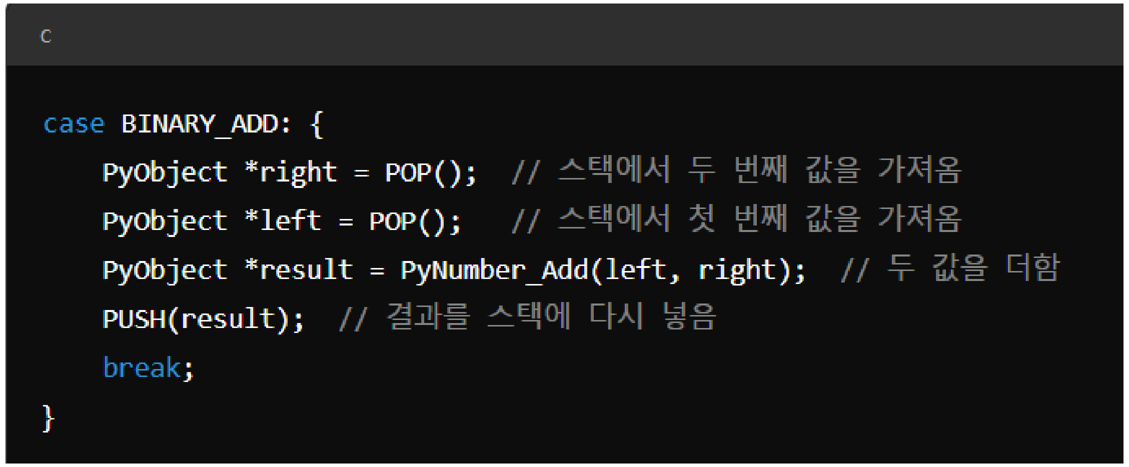
- Key Difference in Interpreter-Based Programs:
  - No Predefined Data or Code Segments: Unlike compiled languages, interpreter-based programs like Python do not allocate memory in predefined data and code segments for the user program. Instead, everything (objects, variables, bytecode) is managed dynamically by the interpreter in heap memory.
  - Interpreter's Role: The interpreter (like CPython for Python) is a compiled program that resides in the traditional code segment, but the Python code itself (bytecode and objects) resides in the heap.
- The interpreter as a Program and Python Bytecode as Data
  - 인터프리터는 code 영역에 로드되고, 파이썬 코드는 바로 실행되는 것이 아니라 데이터로서 다루어진다..
  - 인터프리터가 파이썬 코드를 Bytecode로 번역하고, 이 Bytecode는 heap 영역에 저장된다. 이때 객체들은 heap에 저장되고, 변수에 대한 참조는 stack에 저장된다. (Python bytecodes are treated as data.)
  - 인터프리터는 Bytecode를 하나씩 읽어 해당하는 내부 함수(미리 정의된 코드 루틴)를 실행하여, Bytecode의 각 명령을 해석하고 수행한다 (즉, 시뮬레이션한다).

Python

- Example Bytecodes ('PyEval_EvalFrameEx' 함수에서 처리)
  - LOAD_CONST: 상수를 stack에 로드
  - BINARY_ADD: 두 값을 stack 에서 꺼내 더한 후 다시 stack 에 저장
  - CALL_FUNCTION: 함수 호출
  - RETURN_VALUE: 함수 결과 반환



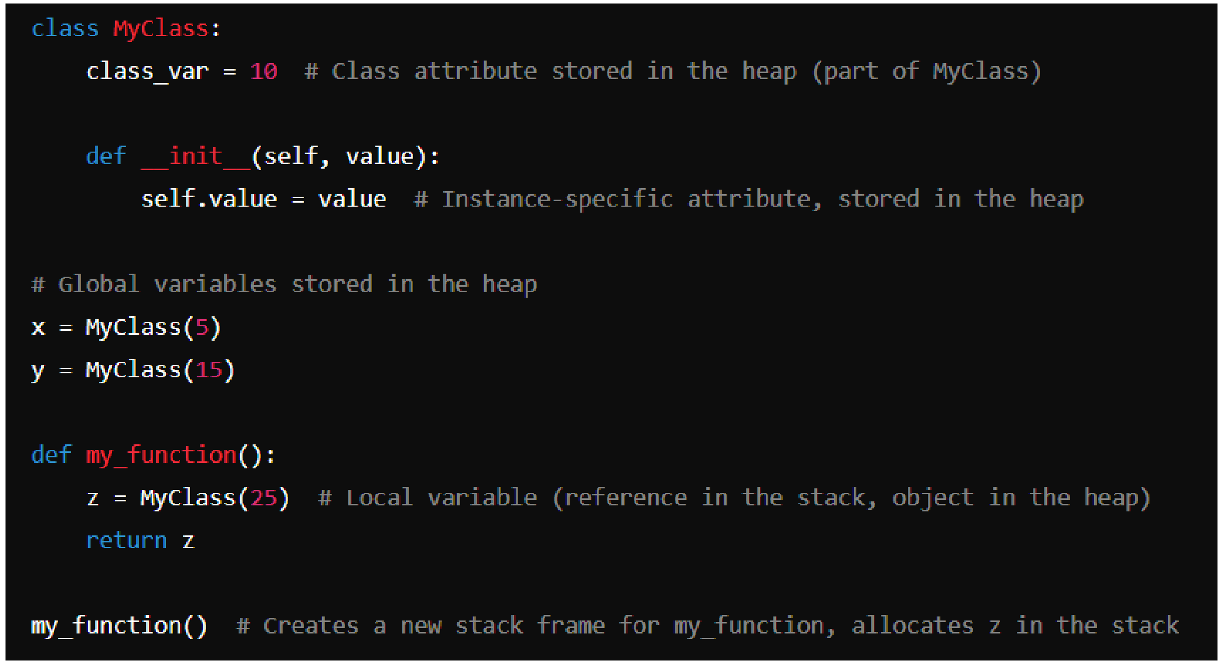
```c
while (bytecode_instruction = *instruction_pointer++) {
    switch (bytecode_instruction) {
        case LOAD_CONST:
            // LOAD_CONST에 해당하는 C 코드 처리 (상수를 스택에 로드)
            break;
        case BINARY_ADD:
            // BINARY_ADD에 해당하는 C 코드 처리 (덧셈 연산)
            break;
        case CALL_FUNCTION:
            // CALL_FUNCTION에 해당하는 C 코드 처리 (함수 호출)
            break;
        // 기타 다른 bytecode 처리...
    }
}
```



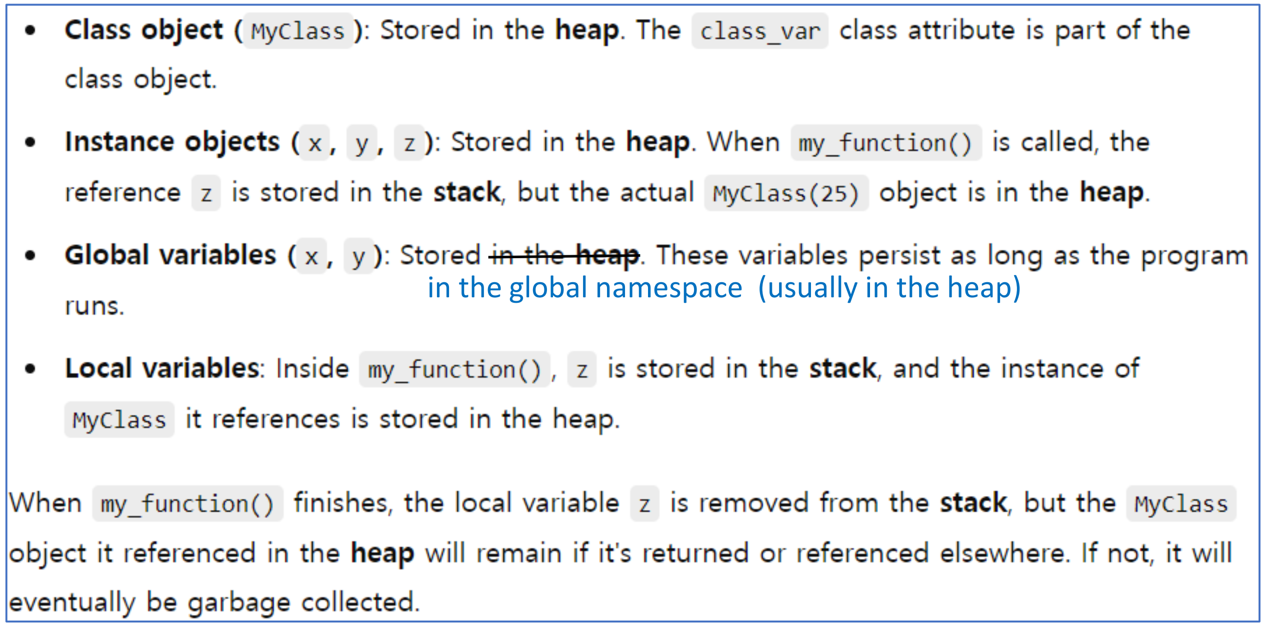
```c
case BINARY_ADD: {
    PyObject *right = POP();  // 스택에서 두 번째 값을 가져옴
    PyObject *left = POP();   // 스택에서 첫 번째 값을 가져옴
    PyObject *result = PyNumber_Add(left, right);  // 두 값을 더함
    PUSH(result);  // 결과를 스택에 다시 넣음
    break;
}
```

PyEval_EvalFrameEx 함수
(CPython 안에서bytecode를
해석하고 실행하는 핵심 함수)

Python Class



```python
class MyClass:
    class_var = 10  # Class attribute stored in the heap (part of MyClass)

    def __init__(self, value):
        self.value = value  # Instance-specific attribute, stored in the heap

# Global variables stored in the heap
x = MyClass(5)
y = MyClass(15)

def my_function():
    z = MyClass(25)  # Local variable (reference in the stack, object in the heap)
    return z

my_function()  # Creates a new stack frame for my_function, allocates z in the stack
```



- **Class object (`MyClass`)**: Stored in the **heap**. The `class_var` class attribute is part of the class object.
- **Instance objects (`x`, `y`, `z`)**: Stored in the **heap**. When `my_function()` is called, the reference `z` is stored in the **stack**, but the actual `MyClass(25)` object is in the **heap**.
- **Global variables (`x`, `y`)**: Stored ~~in the **heap**~~ in the global namespace (usually in the heap). These variables persist as long as the program runs.
- **Local variables**: Inside `my_function()`, `z` is stored in the **stack**, and the instance of `MyClass` it references is stored in the heap.

When `my_function()` finishes, the local variable `z` is removed from the **stack**, but the `MyClass` object it referenced in the **heap** will remain if it's returned or referenced elsewhere. If not, it will eventually be garbage collected.

blank

Python Language

- What is Python?
  - Widely used general purpose high-level language
  - There are two major versions : Python 2 and Python 3
  - Object-oriented
  - Interpreted language
  - Two modes: Interactive Mode and Script Mode
- Important Concepts
  - Objects, attributes, and methods
  - Functions vs. object methods
  - Object references
  - Mutable and immutable objects

Python Programs

- a program is a sequence of definitions and commands
  - definitions evaluated
  - commands executed by Python interpreter in a shell
- commands (statements) instruct interpreter to do something
- can be typed directly in a shell or stored in a file that is read into the shell and evaluated
- Programs manipulate data objects, and Objects are:
  - Scalar (can not be subdivided): basic type
  - non-Scalar (have internal structure that can be accessed): container type

Data types

- **Basic Types (Scalar Objects)**
  - Int (정수): 크기 제한이 없는 정수, 100, -5 등
  - Float (실수): 소수점이 있는 숫자 (64비트). 3.14, -0.01 등
  - Bool (불리언): 논리값, True or False
  - NoneType (special and has one value): 값이 없음을 나타내는 특별한 값. None
- **Container types: 여러 데이터를 하나의 묶음으로 담는 자료구조**
  - Str (스트링): sequence of characters, “Hello”
  - List (리스트): can contain any types of variables, mutable, [1, 2.3, ”Welcome”]
  - Tuple (튜플): can read, but not overwrite (to make computation fast), immutable, (1, 3, [2,3])
  - Dict (딕셔너리): only access by keys, mutable, {“name”:”Kim”, “age”:25}

Data types

- **Array (or ndarray)**
  - Defined in numpy: similar to a list, but much more efficient
  - all the elements are of the same type (int, float, Boolean, string, or other object)
  - Element-wise operation (vector operation)
- **DataFrame and Series**
  - Defined in pandas: provides data processing and analysis capabilities
  - Built on top of ‘numpy’ functionality
  - Table-shaped: “columns” and “index”

Binding variables and values

- Equal sign is an **assignment** of a value to a variable name

  pi = 3.14159

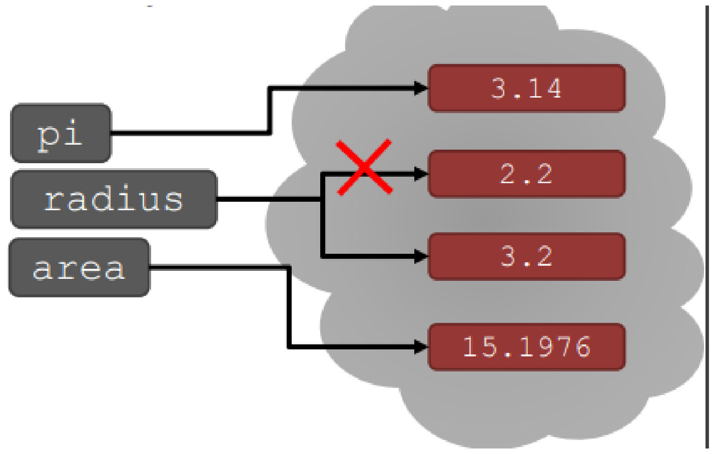
- **Assignment** of expression:

  radius = radius + 1

  - expression on the right (evaluated to a value)
  - variable name on the left
  - Equivalent to `radius += 1`

- Can **re-bind** variable names using new assignments

```
pi = 3.14
radius = 2.2
area = pi*(radius**2)
radius = radius+1
```



Objects, attributes, and methods

- **Everything in Python is an object**.
  - Scalars, sequences, dictionaries, functions, DataFrames, modules, and more
  - Object is simply a collection of data (variables) and methods (functions) that act on those data.
- Each type of object has a set of
  - **Attributes**: Characteristics of the object
  - **Methods**: Functions that operate on the object (and possibly other objects)
- Attributes and methods are accessible by:
  - obj.attr_name
  - obj.method_name()

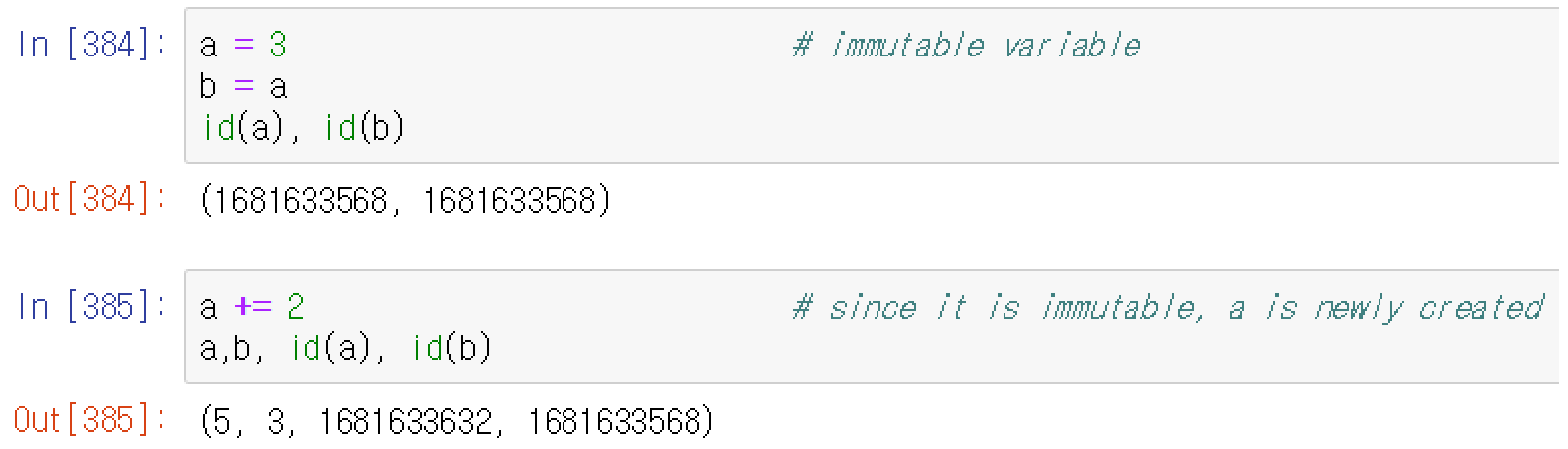
Functions vs. Object Methods

- Functions and object methods are essentially the same…
  - One or more bundled steps performed on some input
  - In some cases, there will be a function and an object method that do the same thing (e.g., sum)
- …BUT, they differ in how they are used
  - Functions are called on zero or more objects and return result(s) that can be assigned to a variable
  - Object methods are called by an object and can either update the calling object or return results

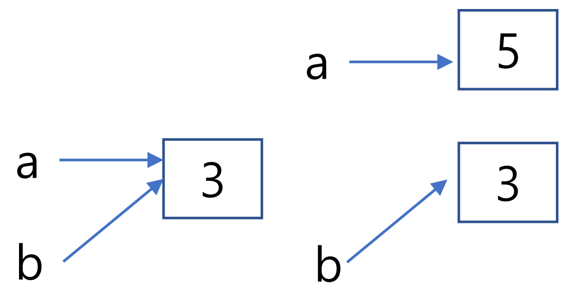
Mutable and Immutable Objects

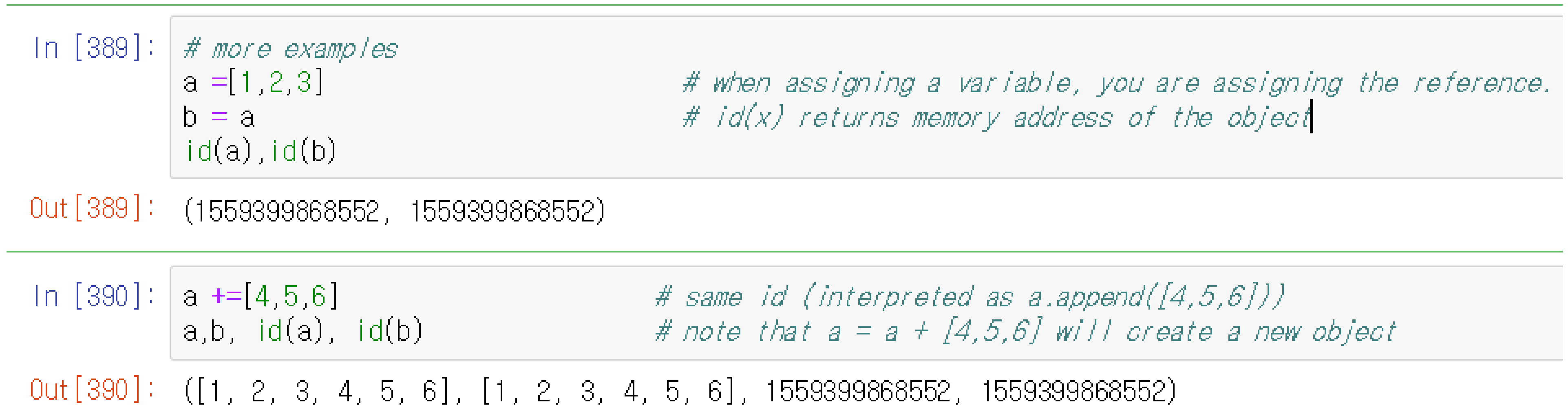
- **Python에서 가장 중요한 개념 중의 하나**
- **Mutable Objects (변경 가능 객체)**
  - 생성된 후에도 내용(요소)을 변경할 수 있다. (in-place 변경)
  - Sets, Lists, dictionaries, arrays, dataframes, class instances
  - (ex) my_list.append(5) 를 실행하면 my_list 가 가리키는 원본 객체가 수정됨.
- **Immutable Objects (변경 불가능 객체)**
  - 생성된 후에는 값을 변경할 수 없다.
  - All other types including int, float, Boolean, strings, tuples
  - (ex) a=10 다음에 a=11 을 실행하면 10 이 11 로 바뀌는 게 아니라 11 이라는 새로운 객체가 생성되고 a 가 그 새 객체를 가리킨다.

Mutable and Immutable (examples)

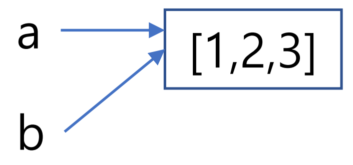


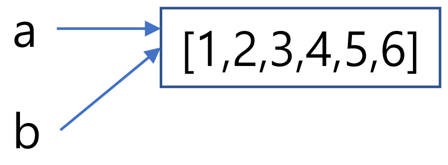
```
In [384]: a = 3                                   # immutable variable
          b = a
          id(a), id(b)

Out[384]: (1681633568, 1681633568)

In [385]: a += 2                                  # since it is immutable, a is newly created
          a,b, id(a), id(b)

Out[385]: (5, 3, 1681633632, 1681633568)
```





```
In [389]: # more examples
          a =[1,2,3]                              # when assigning a variable, you are assigning the reference.
          b = a                                   # id(x) returns memory address of the object
          id(a),id(b)

Out[389]: (1559399868552, 1559399868552)

In [390]: a +=[4,5,6]                             # same id (interpreted as a.append([4,5,6]))
          a,b, id(a), id(b)                       # note that a = a + [4,5,6] will create a new object

Out[390]: ([1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6], 1559399868552, 1559399868552)
```





How Functions Really Work: Call by Object Reference

- "왜 함수 안에서 리스트는 바뀌고, 숫자는 안 바뀔까?"
- 파이썬은 **'Call by Object Reference'** 방식을 사용
  - 함수에 인자를 넘길 때, 해당 객체의 참조(메모리 주소)가 함수 안의 매개변수로 복사
- 동작 방식:
  - Immutable 인자 전달: 함수 안에서 값을 변경하면, 새로운 객체가 생성되고 함수 안의 매개변수가 그 새 객체를 가리킨다. 원본 객체는 그대로 유지된다. (Call-by-Value처럼 동작)
  - Mutable 인자 전달: 함수 안에서 객체의 내용을 변경하면, 함수 안과 밖의 변수가 같은 객체를 가리키고 있으므로 원본 객체가 직접 변경된다. (Call-by-Reference처럼 동작)

Object References

- Call-by-value? or Call-by-reference?

```
>>> def test(a):
        a = 2

>>> a = 1
>>> test(a); a
1
```

```
>>> def test2(a):
        a.append('world.')

>>> b = 'Hello'
>>> test2(b); b
['Hello', 'World.']
```

```
>>> a = 10
>>> b = a
>>> a += 100
>>> a, b
(110, 10)
>>> id(a), id(b)
(14073…7824, 1407…624)
```

```
>>> a = [1,2,3]
>>> b = a
>>> a += [4,5,6]
>>> a,b
([1,2,3,4,5,6],[1,2,3,4,5,6])
>>> id(a), id(b)
(225009…832, 225009…832)
```

Object References (2)

- **Call-by-Object** (or **call-by-Object Reference**)
  - If you pass immutable arguments like integers, strings or tuples to a function, the passing acts like *call-by-value*. The object reference is passed to the function parameters. They can't be changed within the function, because they can't be changed at all, i.e. they are immutable.
  - If mutable arguments are passed, they are also passed by object reference, but they can be changed in place in the function. If we pass a list to a function, we have to consider two cases:
    - Elements of a list can be changed in place, i.e. the list will be changed even in the caller's scope. (act like *call-by-reference*)
    - If a new list is assigned to the name, the old list will not be affected. (a new object is created)

Reusing Your Code: Functions

- **함수 (Function):** 특정 작업을 수행하는 재사용 가능한 코드 블록
  - def 키워드로 정의(define)하고 이름으로 호출 (call)
  - 매개변수(parameter): 함수가 받을 입력 값 (정의 시)
  - 인자(argument): 함수에 실제로 전달하는 값 (호출 시)
  - Return 키워드로 결과값 반환
- **내장 함수 (Built-in) vs. 사용자 정의 함수 (User-Defined)**
  - 내장함수(Built-in Functions): print(), len(), sum() 처럼 파이썬이 기본 제공하는 함수
  - 사용자 정의 함수: 사용자가 def 으로 직접 만드는 함수

Organizing Your Code: Modules & Packages

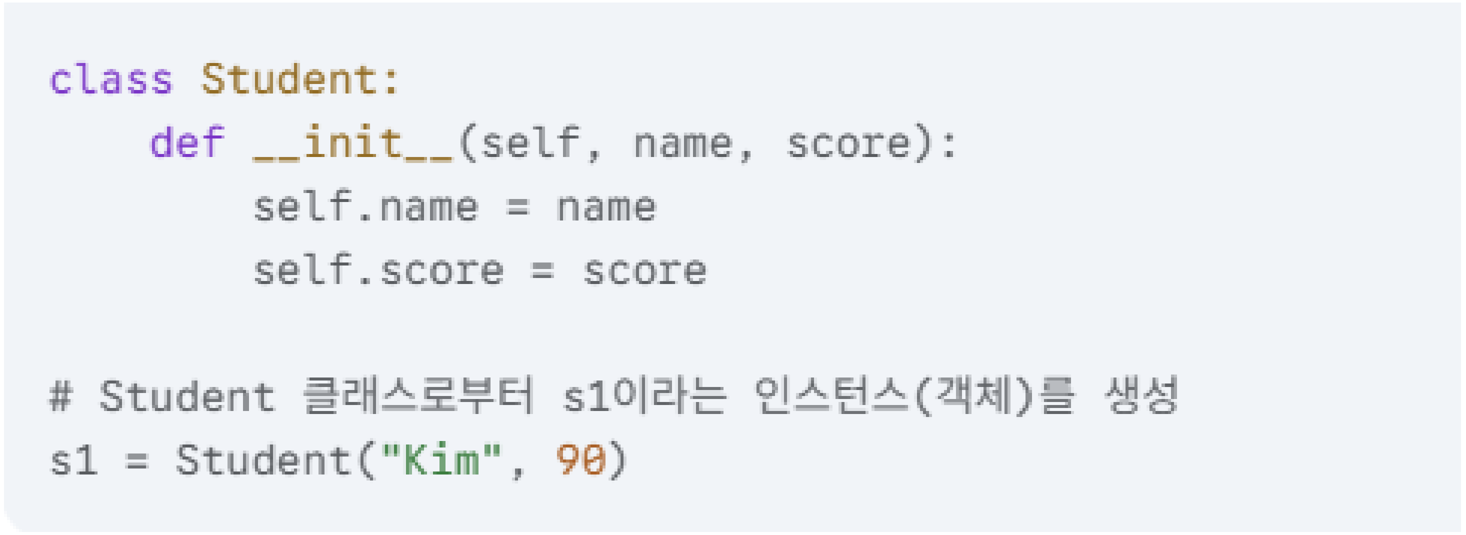
- **모듈 (Module):** 함수, 클래스, 변수들을 모아놓은 하나의 파이썬 파일 (.py)
- **패키지 (Package):** 여러 관련 모듈들을 모아놓은 디렉터리(폴더)
  - 이 디렉터리가 패키지임을 나타내는 __init__.py 파일이 반드시 포함되어야 함
- 비유:
  - 모듈: 책의 '한 챕터'
  - 패키지: 여러 챕터가 모인 '책 한 권'
  - 라이브러리: 여러 책이 모인 '도서관'

Importing Modules and Scripts

- Modules and Python scripts are loaded in the same manner. For a module or Python script P (.py):
  - `import P [as p]`
  - `from module_name import *` // import all functionality
  - `from module_name import f, g, h` // import specific functions
  - `Import foo.bar (or from foo import bar)` // import module bar from package foo
- **Built-in modules (standard library)**
  - https://docs.python.org/3/library/

Object-Oriented Programming (OOP) in Python

- 파이썬의 모든 것은 객체(Object)이다.
- **클래스 (Class):** 객체를 만들기 위한 설계도 또는 틀.
  - **속성 (Attributes):** 객체가 가지는 데이터 (예: 학생의 이름, 점수)
  - **메서드 (Methods):** 객체가 수행할 수 있는 동작 (예: 학생 정보 출력)
- **객체 (Object) / 인스턴스 (Instance):** 클래스라는 설계도로부터 만들어진 실체
- (예)



```python
class Student:
    def __init__(self, name, score):
        self.name = name
        self.score = score

# Student 클래스로부터 s1이라는 인스턴스(객체)를 생성
s1 = Student("Kim", 90)
```

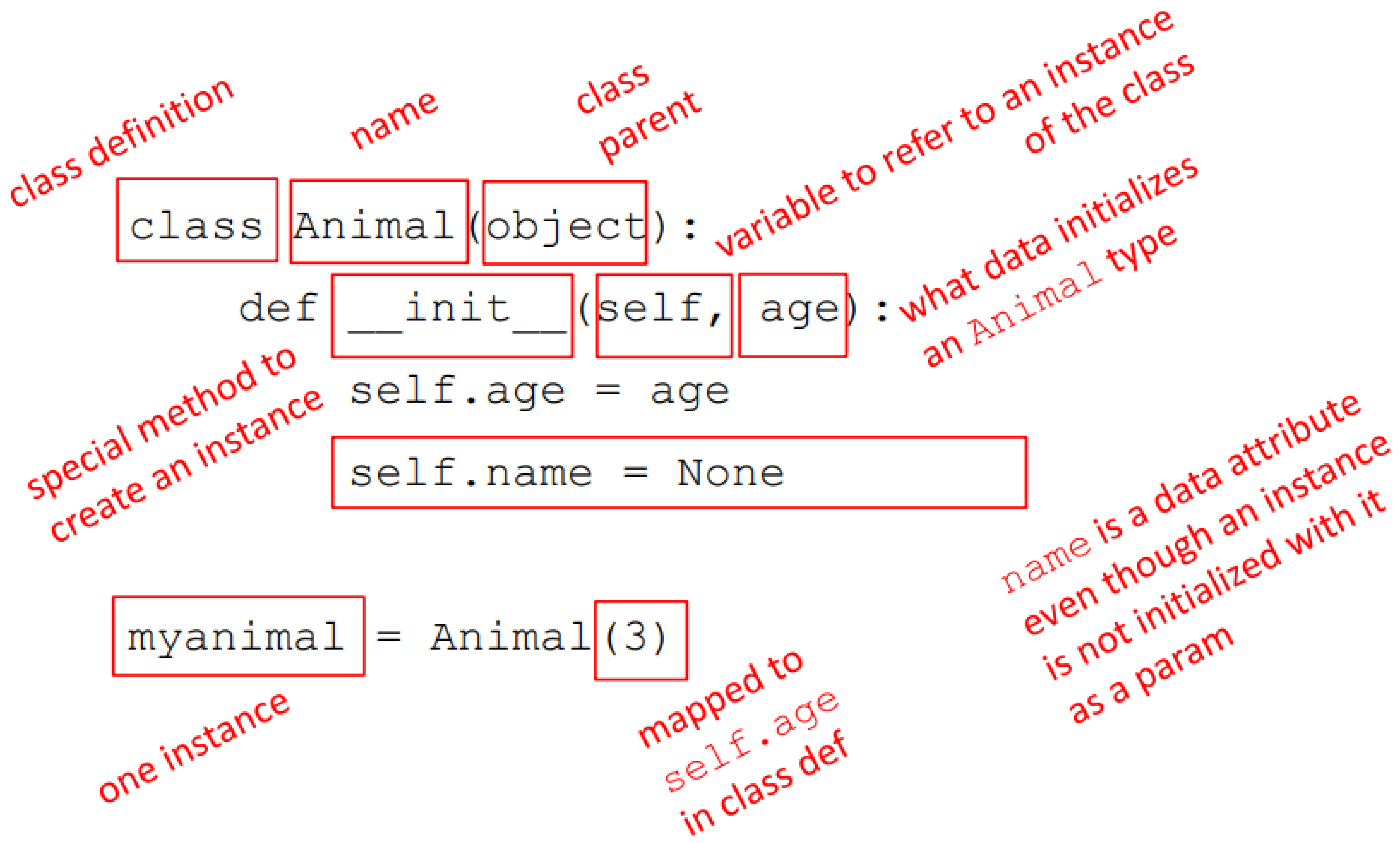
Classes and Objects

- **Implementing a new object type with classes**
  - **define** the class
  - define **data attributes** and **methods**
  - data and methods are **common across all instances**

- **Using the new object type**
  - create **instance** of the object type (instance is **one specific** object)
  - do **operations**
  - Instance has the **structure of the class**

- Objects have attributes
  - Data attributes : how can you represent your object with data? (**what it is**)
  - Procedural attributes (behavior/operations/methods): how can someone interact with the object (**what it does**)
  - *'self'* is a special parameter referring to an instance of the class

Classes define example



```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None

myanimal = Animal(3)
```

class definition
name
class parent
variable to refer to an instance of the class
what data initializes an Animal type
special method to create an instance
name is a data attribute even though an instance is not initialized with it as a param
one instance
mapped to self.age in class def

Indexing and Slicing

- for container variables: lists, arrays, tuples, and strings
  - e.g., A = [1,2,3]
- **Indexing**: access item in a sequence
  - Python is zero-based: A[0] = 1, A[1] = 2, A[2] = 3
  - Negative indices: A[-1] = 3, A[-2] = 2
- **Slicing**: access subset of a sequence [*start: stop: step*]
  - A[start=0: stop=len(A): step=1]
  - Slicing ends before the *stop* (excluding *stop*): A[0:1] = [1]; A[0:2] = [1,2]
- Examples
  - A[:] = A[::] = A = [1,2.3]
  - A[1:] = [2,3], A[:2] =[1,2]
  - A[::2] = [1,3]
  - A[::-1] = [3,2,1]
  - A[-1:0:-1] = [3,2]
  - More…

The NumPy

- Foundation for scientific computing
  - Linear algebra, math functions, and random number generation
- Provides useful data structures (array)
  - Ndarray (or array) : similar to lists, but much more powerful
  - Vectorization: fast operations on arrays of data without the need for loops
- Primary Use:
  - Fast vectorized array operations for data munging, cleaning, filtering, transforming
  - Built-in common array algorithms
  - Efficient descriptive statistics
  - Data alignment and relational data manipulations for merging and joining multiple data sets
  - Expressing conditional logic ad array expressions instead of loops

NumPy - array

- Numerical Python N-dimensional arrays
  - Similar to list, but much more powerful
- Fast, flexible container for data
  - Numerical, boolean, or string data
- Perform mathematical operations on entire data sets without loops (**vectorization**)
- Some common attributes (i.e., arr.*attr*)
  - .ndim: Number of array dimensions (e.g., 1, 2, 3)
  - .shape: Size of each dimension (e.g., (2, 4))
  - .dtype: Data type for the array (e.g., int8, float64)

Slicing: list and array

- 1-D array slicing (quite often used)

```
a = np.arrange(10)    # a = array([0,1,2,3,4,5,6,7,8,9])
a[start:end]          # items start through end-1
a[start:]             # items start through the rest of the array
a[:end]               # items from the beginning through end-1
a[:]                  # a copy of the whole array
a[start:end:step]     # start through not past end, by step

a[-1]                # last item in the array
a[-2:]               # last two items in the array
a[:-2]               # everything except the last two item
a[::-1]              # all items in the array, reversed
a[1::-1]             # the first two items, reversed
a[:-3:-1]            # the last two items, reversed
a[-3::-1]            # everything except the last two items, reversed
```

Slicing: list and array

- 2-D array slicing (to split loaded data into input(X) and the output(y))

```
X = [:, :-1]   # select al the rows and all columns except the last one
y = [:, -1]    # select all rows again, and index just the last column
```

Array operations

- Between Arrays and Scalars -- Broadcasting
  - All basic operations are applied **element-wise**
  - +, -, /, *, **, %, etc.
- Universal Functions (ufunc)
  - Unary (on a single array): abs, sqrt, exp, log, ceil, floor, logical_not, and more
  - Binary (on two equal-sized arrays): +, -, /, *, **, min, max, mod, >, >=, <, <=, ==, !=, &, |, ^
- Mathematical and Statistical Functions/Methods
  - Aggregation (collection): mean(), sum(), std(), var(), min(), max(), argmin(), argmax()
  - Non-aggregation: cumsum(), cumprod()
  - Sort, concatenate, etc.

Pandas

- Pandas
  - 파이썬을 위한 고성능 데이터 분석 라이브러리
- Pandas의 역할:
  - 표 형식(Tabular) 데이터를 다루는 데 최적화되어 있다.
  - 데이터 로딩, 정제, 처리, 필터링, 그룹화, 병합 등 데이터 분석의 전 과정을 지원.
  - NumPy를 기반으로 만들어져 강력한 성능.
- 두 가지 핵심 자료구조: Series, DataFrame
- 주요 기능:
  - Creating Series and DataFrame objects
  - Basic Series and DataFrame methods
  - Indexing/reindexing, slicing, and filtering
  - Mathematical operations
  - Missing data handling

Series and DataFrame

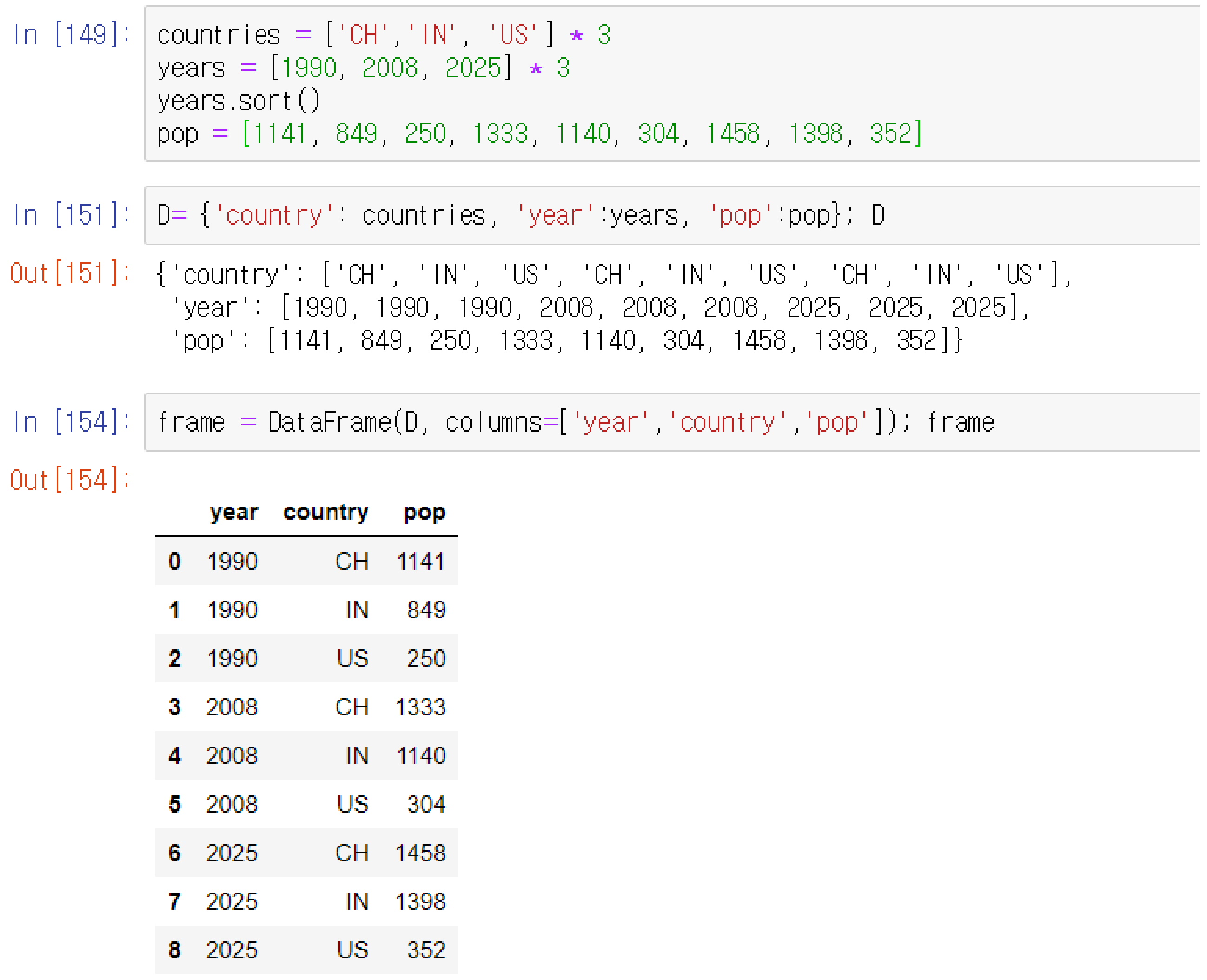
- Series:
  - 1차원 배열과 같은 구조로, 하나의 데이터 열(column)을 나타
  - 각 데이터에 접근할 수 있는 인덱스(index)를 가진다.
  - A single column or a row of a DataFrame
- DataFrame:
  - 2차원 테이블(표) 형태의 자료구조
  - 여러 개의 Series 가 모여 하나의 DataFrame 을 이룬다.
  - 행(row)과 열(column)에 각각 인덱스를 가진다.
  - 데이터 분석 작업의 가장 기본이 되는 데이터 구조
  - Access rows by **index**, and columns by **column names**
  - Built-in methods for data processing, computation, visualization, and aggregation

Essential Pandas Operations

- 데이터 생성: 딕셔너리나 NumPy 배열로부터 DataFrame을 쉽게 구성.
- 데이터 선택 (Indexing and Slicing)
    - `df['column_name']`: 특정 열 선택
    - `df.loc[]`: 레이블(이름) 기반으로 행/열 선택
    - `df.iloc[]`: 숫자 위치 기반으로 행/열 선택
- 기본 메서드:
    - `.head()`: 처음 5개 행 보기
    - `.info()`: 데이터프레임의 요약 정보(데이터 타입, null 값 등) 확인
    - `.describe()`: 숫자형 데이터의 기술 통계량 요약
    - `.sort_values()`: 특정 열을 기준으로 데이터 정렬

Pandas – DataFrames (example)

- From dictionary



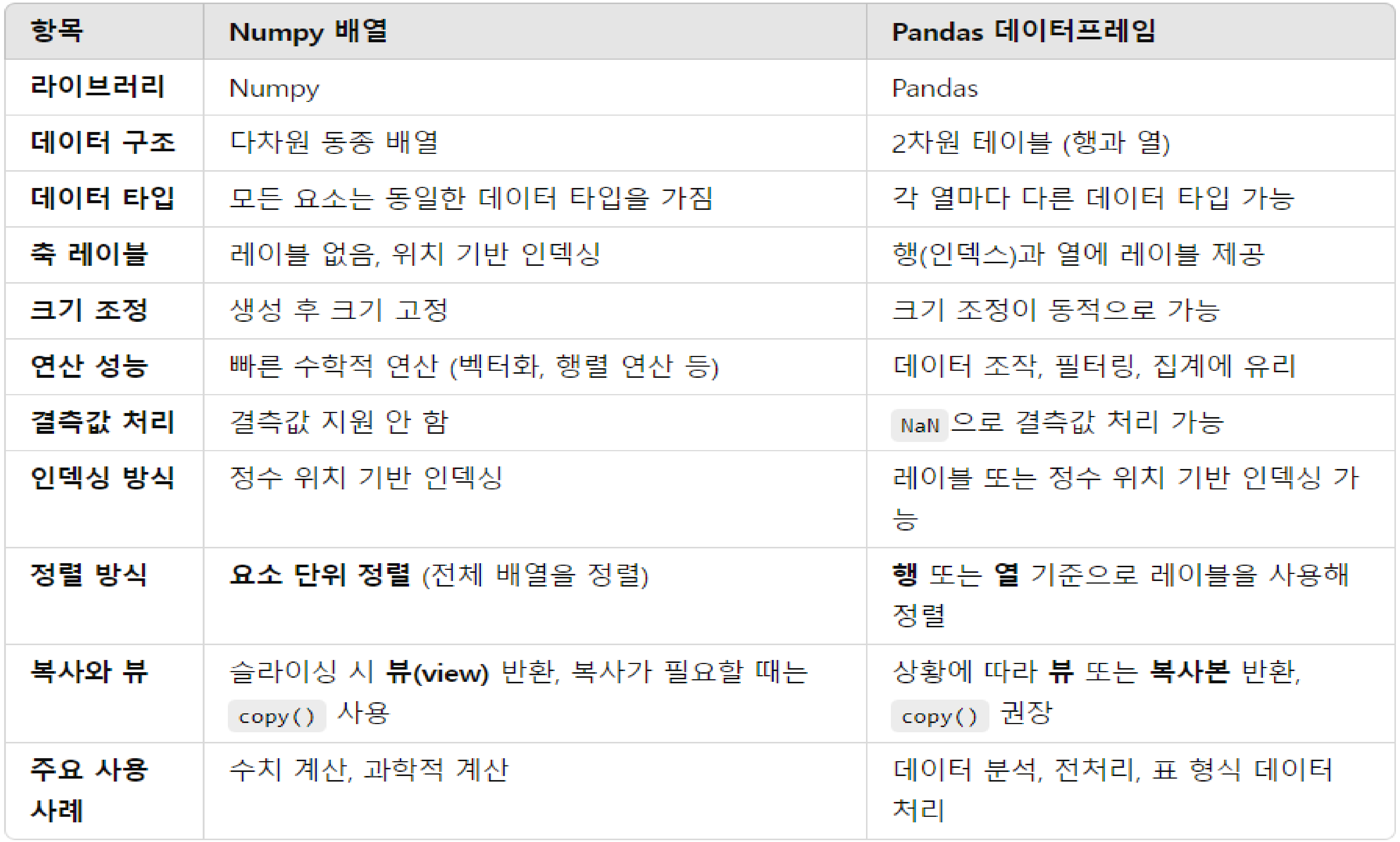
```
In [149]: countries = ['CH','IN', 'US'] * 3
          years = [1990, 2008, 2025] * 3
          years.sort()
          pop = [1141, 849, 250, 1333, 1140, 304, 1458, 1398, 352]

In [151]: D= {'country': countries, 'year':years, 'pop':pop}; D

Out[151]: {'country': ['CH', 'IN', 'US', 'CH', 'IN', 'US', 'CH', 'IN', 'US'],
           'year': [1990, 1990, 1990, 2008, 2008, 2008, 2025, 2025, 2025],
           'pop': [1141, 849, 250, 1333, 1140, 304, 1458, 1398, 352]}

In [154]: frame = DataFrame(D, columns=['year','country','pop']); frame

Out[154]:
```

	year	country	pop
0	1990	CH	1141
1	1990	IN	849
2	1990	US	250
3	2008	CH	1333
4	2008	IN	1140
5	2008	US	304
6	2025	CH	1458
7	2025	IN	1398
8	2025	US	352

Pandas - DataFrames

- Basic DataFrame Methods
  - Indexing columns(features): either by column name or attribute (ex: df[‘year’] or df.year, df[[‘year’,’pop’]])
  - Indexing rows by index name or index number: df.loc() or df.iloc()
  - df.name, df.index, df.columns, and df.values (similar to Series)
- Functions
  - df.sort_index(), df.sort_index(axis=1) // sort by index or columns
  - df.sum(), df.mean()
  - df.idmax(), df.idmin()   // index of max and min
  - df.value_counts()   // counts of values
  - df.isin([‘b’,’c’])   // see if some elements are in df
  - df.fillna(), df.dropna() // remove or fill any columns of NaN

Array and DataFrame



항목	Numpy 배열	Pandas 데이터프레임
라이브러리	Numpy	Pandas
데이터 구조	다차원 동종 배열	2차원 테이블 (행과 열)
데이터 타입	모든 요소는 동일한 데이터 타입을 가짐	각 열마다 다른 데이터 타입 가능
축 레이블	레이블 없음, 위치 기반 인덱싱	행(인덱스)과 열에 레이블 제공
크기 조정	생성 후 크기 고정	크기 조정이 동적으로 가능
연산 성능	빠른 수학적 연산 (벡터화, 행렬 연산 등)	데이터 조작, 필터링, 집계에 유리
결측값 처리	결측값 지원 안 함	`NaN` 으로 결측값 처리 가능
인덱싱 방식	정수 위치 기반 인덱싱	레이블 또는 정수 위치 기반 인덱싱 가능
정렬 방식	**요소 단위 정렬** (전체 배열을 정렬)	**행** 또는 **열** 기준으로 레이블을 사용해 정렬
복사와 뷰	슬라이싱 시 **뷰(view)** 반환, 복사가 필요할 때는 `copy()` 사용	상황에 따라 **뷰** 또는 **복사본** 반환, `copy()` 권장
주요 사용 사례	수치 계산, 과학적 계산	데이터 분석, 전처리, 표 형식 데이터 처리

(*) 리스트의 슬라이스는 copy(복사)

blank

Data Visualization - Matplotlib

- Use:
  - %matplotlib inline magic command (once Jupyter is open)
  - import matplotlib.pyplot as plt
- Basic template
  - Create a new figure : (ex) fig = plt.figure(figsize = (12,8))
  - Add subplots (if necessary)
    - ax1 = fig.add_subplot(2,1,1)   # 2x1 arrangement, first figure
    - ax2 = fig.add_subplot(2,1,2)
  - Create plot (plt or ax1…axN methods)
  - Label, annotate, format plot
  - Copy or save plot

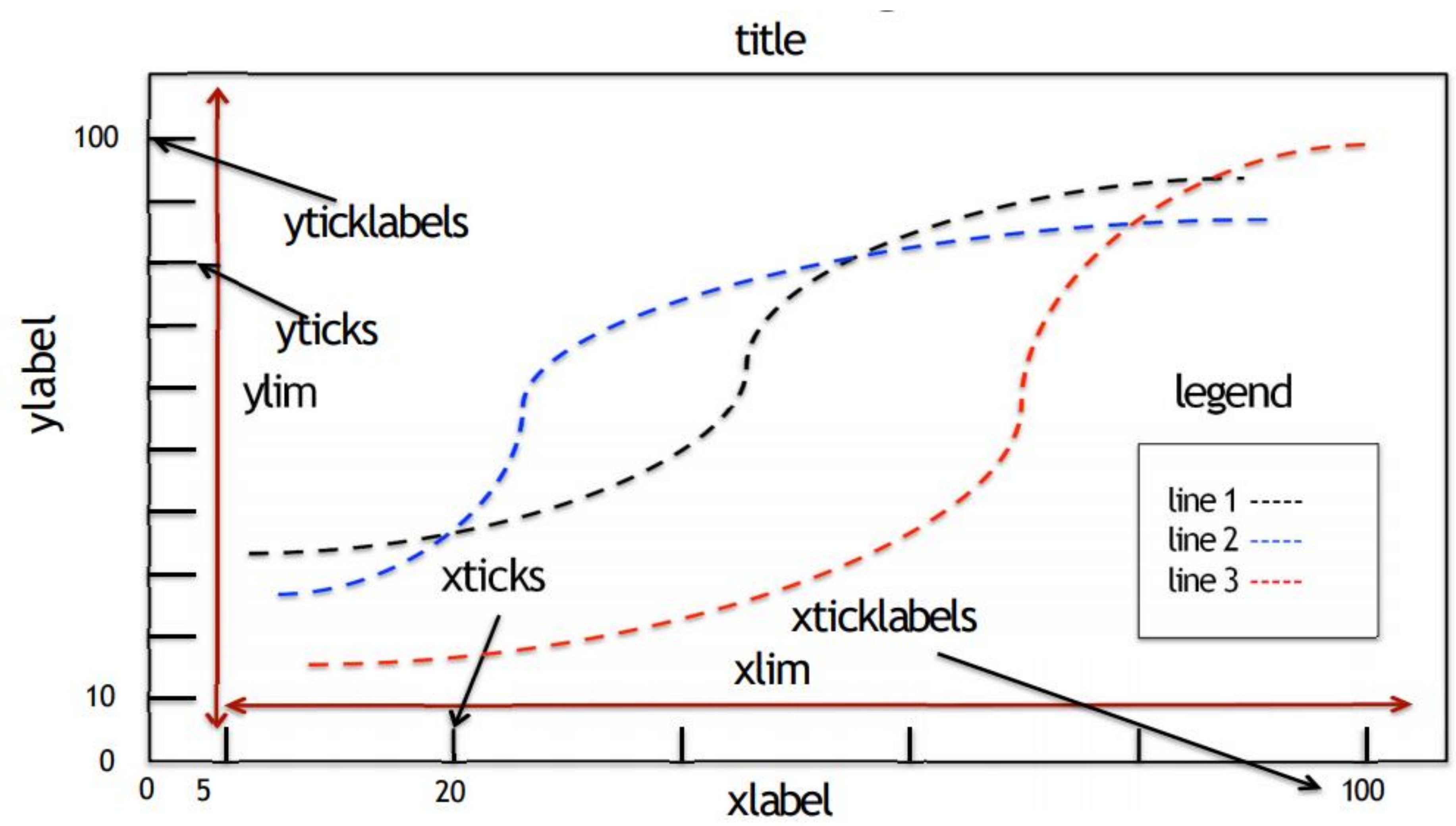
Matplotlib - Common plot types

- Line plots
  - `plt.plot (x, y, ‘-’)`
- Scatter plots – comparison between lots of data
  - `plt.scatter (x, y, ‘.’)`
- Bar plots – comparison between few data
  - Bar (horizontal): `plt.barh (x, y, width)`
  - Column (vertical): `plt.bar (x, y, width)`
- Histogram plots – single distributions
  - `plt.hist (x, bins)`
- Boxplots – one or more distributions
  - `plt.boxplot (x)`

Matplotlib - Colors, Markers, and Line Styles

- All specified as special string characters in plot call
- Colors - Many plot types
  - Basic colors: g(reen), r(ed), b(lue), (blac)k, m(agenta), y(ellow), c(yan), w(hite)
  - For more, see http://matplotlib.org/api/colors_api.html
- Markers and Line Styles - Mostly relate to plt.plot
  - Markers: ., o, +, * (star), 1, 2, 3, 4 (triangles), s(quare), D(iamond)
  - Line styles: solid (-), dashed (--), dotted (:), dash-dot (-.)
  - linewidth keyword (float value)
- Usage
  - Style string: Combines all three (e.g., ‘k.’, ‘g--’, ‘ro-’)
  - Separate keyword arguments: color, linestyle, marker

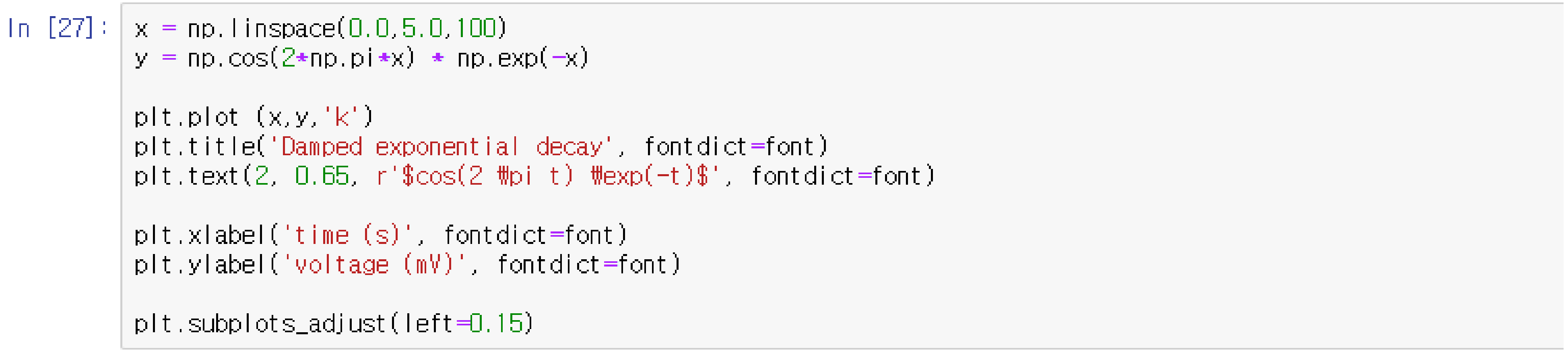
Formatting plots



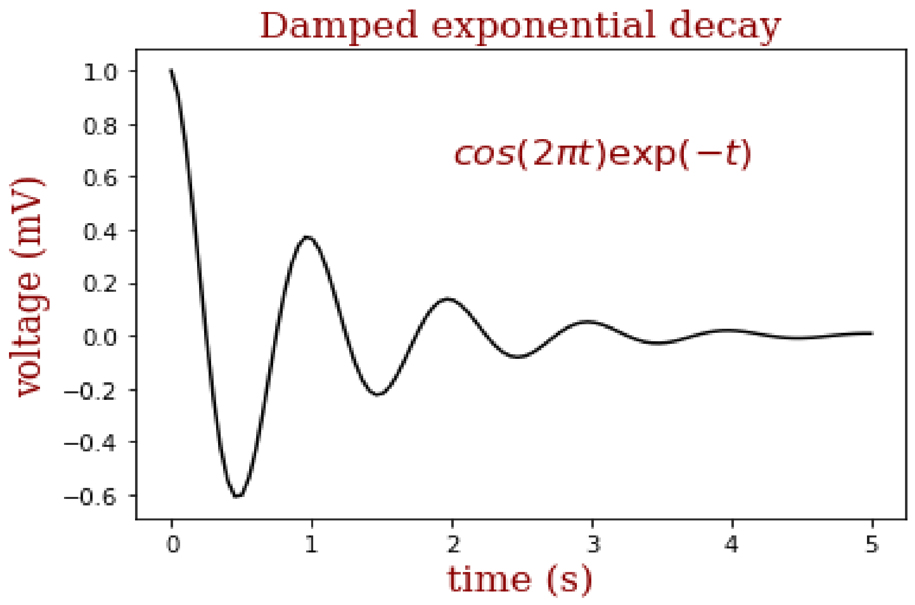
Formatting plots

- Title: title(‘title’)
- Axis labels: xlabel (‘Time’), ylabel (‘Price)
- Axis limits: xlim([0,10]), ylim
- Ticks: xticks([0,60,70,80,90,100]), yticks
- Tick labels: xticklabels([‘F’,’D’,’C’,’B’,’A’]), yticklebals
- Legends: legend((‘one’,’two’,’three’))
- Text
  - text(x, y, text, fontsize)
  - arrow(x, y, dx, dy) # draws arrow from (x,y) to (x+dx, y+dy)
  - annotate (text, xy, xytext) # annotate the xy point with text positioned at xytext
- shapes
  - Rectangles, circles, polygons
  - Location, size, color, transparency (alpha)

Matplotlib - Example(1)



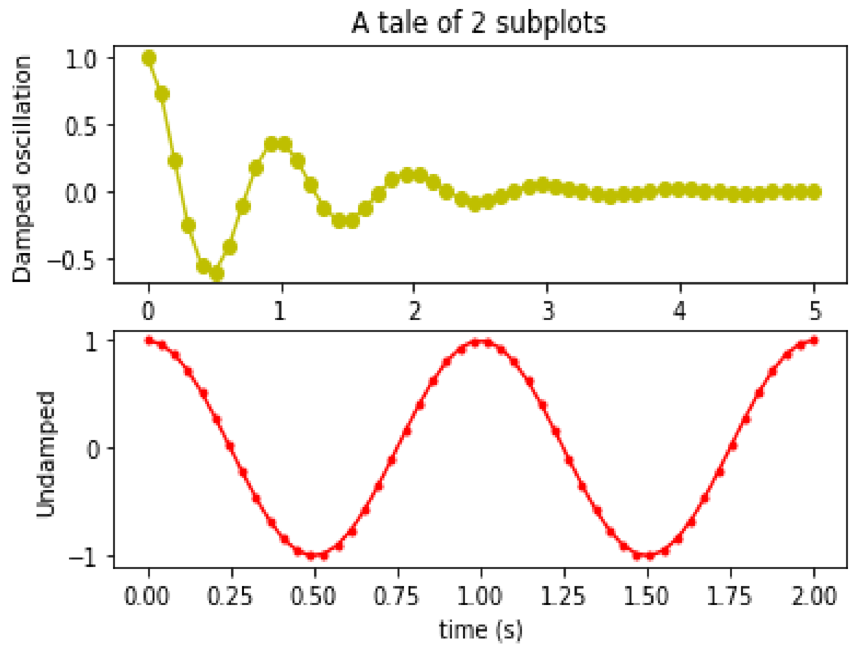
```
In [27]: x = np.linspace(0.0,5.0,100)
         y = np.cos(2*np.pi*x) * np.exp(-x)

         plt.plot (x,y,'k')
         plt.title('Damped exponential decay', fontdict=font)
         plt.text(2, 0.65, r'$cos(2 ₩pi t) ₩exp(-t)$', fontdict=font)

         plt.xlabel('time (s)', fontdict=font)
         plt.ylabel('voltage (mV)', fontdict=font)

         plt.subplots_adjust(left=0.15)
```



Matplotlib - Example(2)

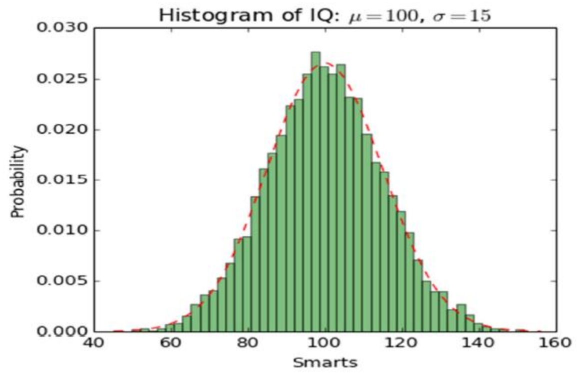
```
In [39]: x1 = np.linspace(0.0,5.0)
         x2 = np.linspace(0.0,2.0)
         y1 = np.cos(2*np.pi*x1) * np.exp(-x1)
         y2 = np.cos(2* np.pi* x2)

         plt.subplot(2, 1, 1)
         plt.plot(x1,y1,'yo-')
         plt.title('A tale of 2 subplots')
         plt.ylabel('Damped oscillation')

         plt.subplot(2, 1, 2)
         plt.plot(x2, y2,'r.-')
         plt.xlabel('time (s)')
         plt.ylabel('Undamped')
```

```
Out[39]: Text(0, 0.5, 'Undamped')
```

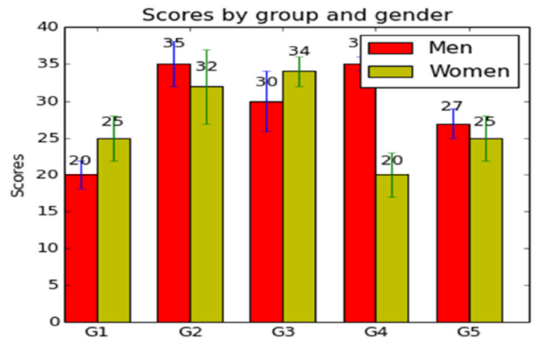


Many more examples...

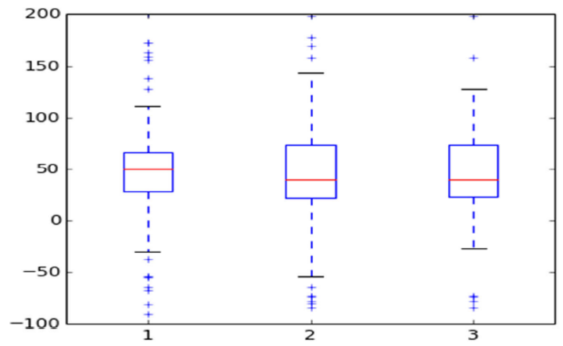
Histogram



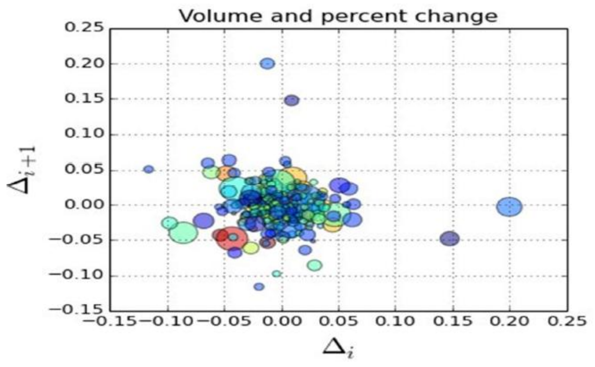
Bar Chart (with error bars and legend)

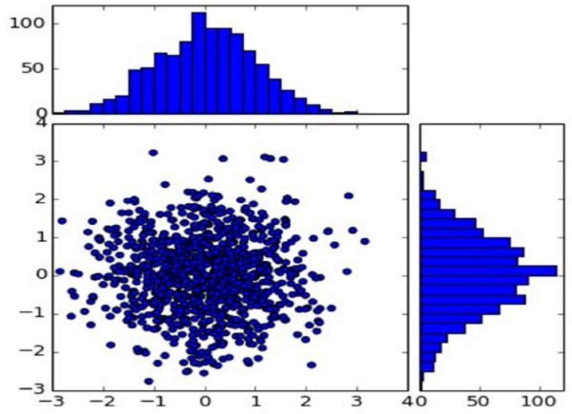


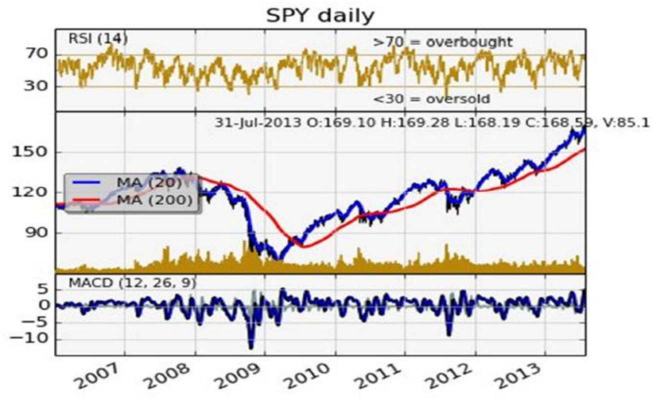
Boxplots



Scatter + Histogram


Volume and percent change
Δ_{i+1}
Δ_i




SPY daily
RSI (14)
>70 = overbought
<30 = oversold
31-Jul-2013 O:169.10 H:169.28 L:168.19 C:168.59, V:85.1
MA (20)
MA (200)
MACD (12, 26, 9)