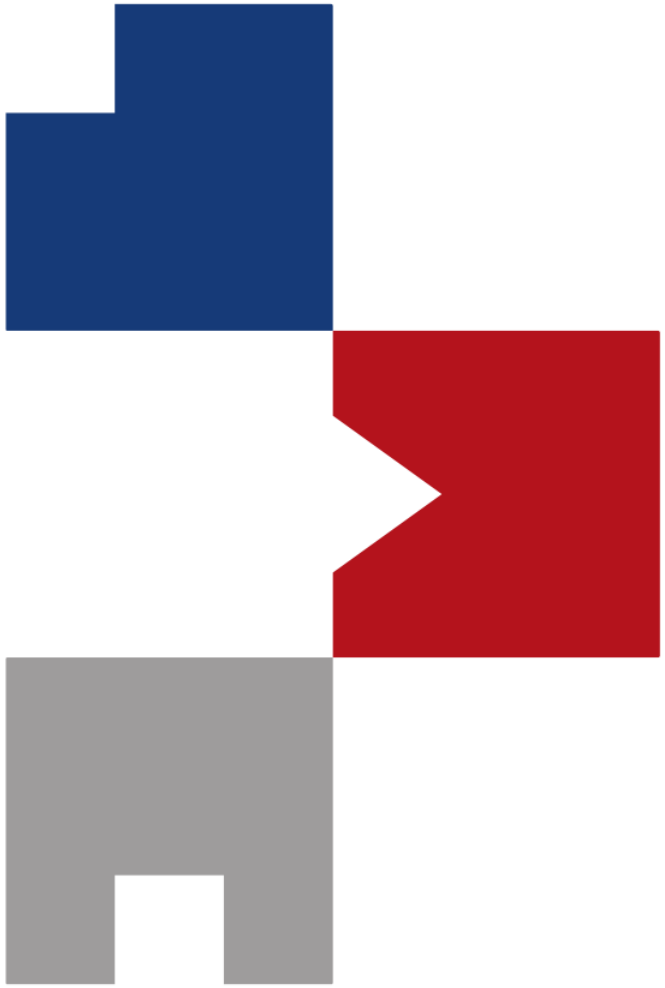


Python: Basic

Sunglok Choi, Assistant Professor, Ph.D.
Computer Science and Engineering Department, SeoulTech
sunglok@seoultech.ac.kr | https://mint-lab.github.io/

Why Python?

- TIOBE Programming Community Index
  - Measure: The number of web pages (counted in Google, MSN, …, YouTube)
  - **Python** is in the second place (on June 24th, 2021).

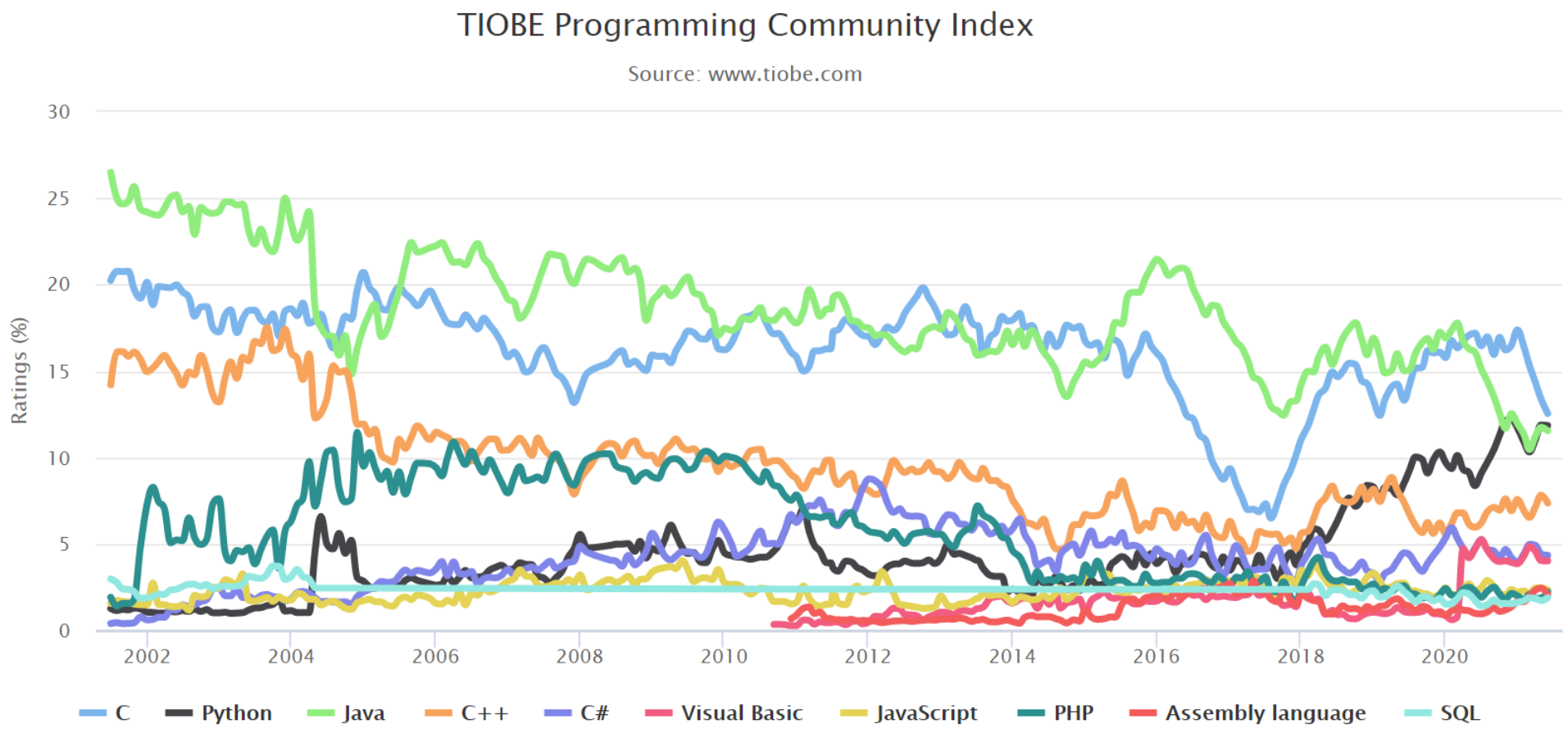


Image: TIOBE Programming Community Index (retrieved on June 24th, 2021)

Why Python?

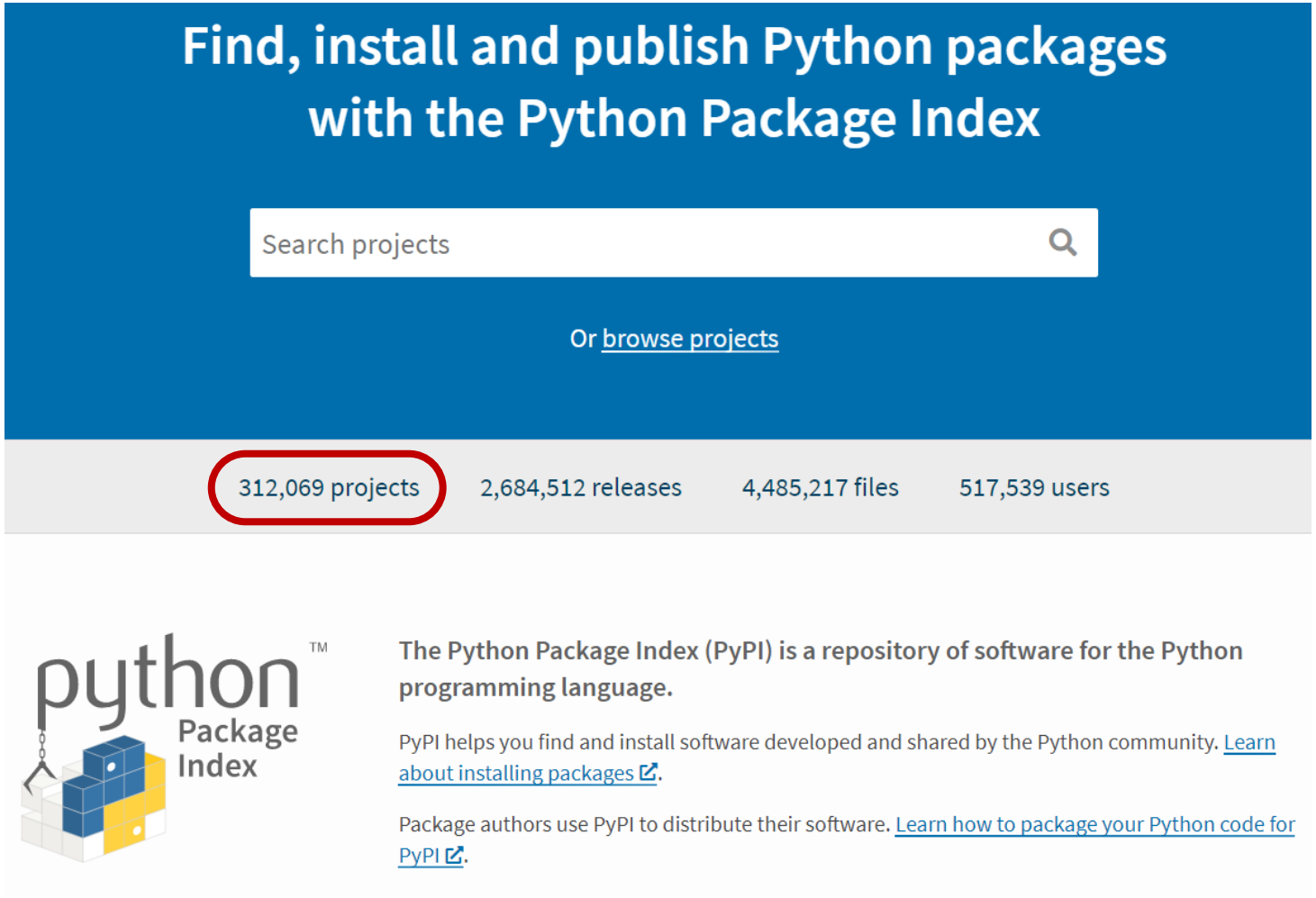
- PYPL (PopularitY of Programming Language)
  - Measure: The number of searches in Google (Google Trends; compared to a year ago)
  - **Python** is in the first place (on June 24th, 2021).

Rank	Change	Language	Share	Trend
1		**Python**	**30.60 %**	**-1.1 %**
2		Java	17.17 %	-0.1 %
3		JavaScript	8.30 %	+0.3 %
4		C#	6.79 %	-0.0 %
5	↑	C/C++	6.31 %	+0.6 %
6	↓	PHP	6.15 %	+0.2 %
7		R	3.88 %	-0.1 %
8		Objective-C	2.54 %	+0.1 %
9	↑	TypeScript	2.10 %	+0.2 %
10	↓	Swift	1.91 %	-0.3 %

Why Python?

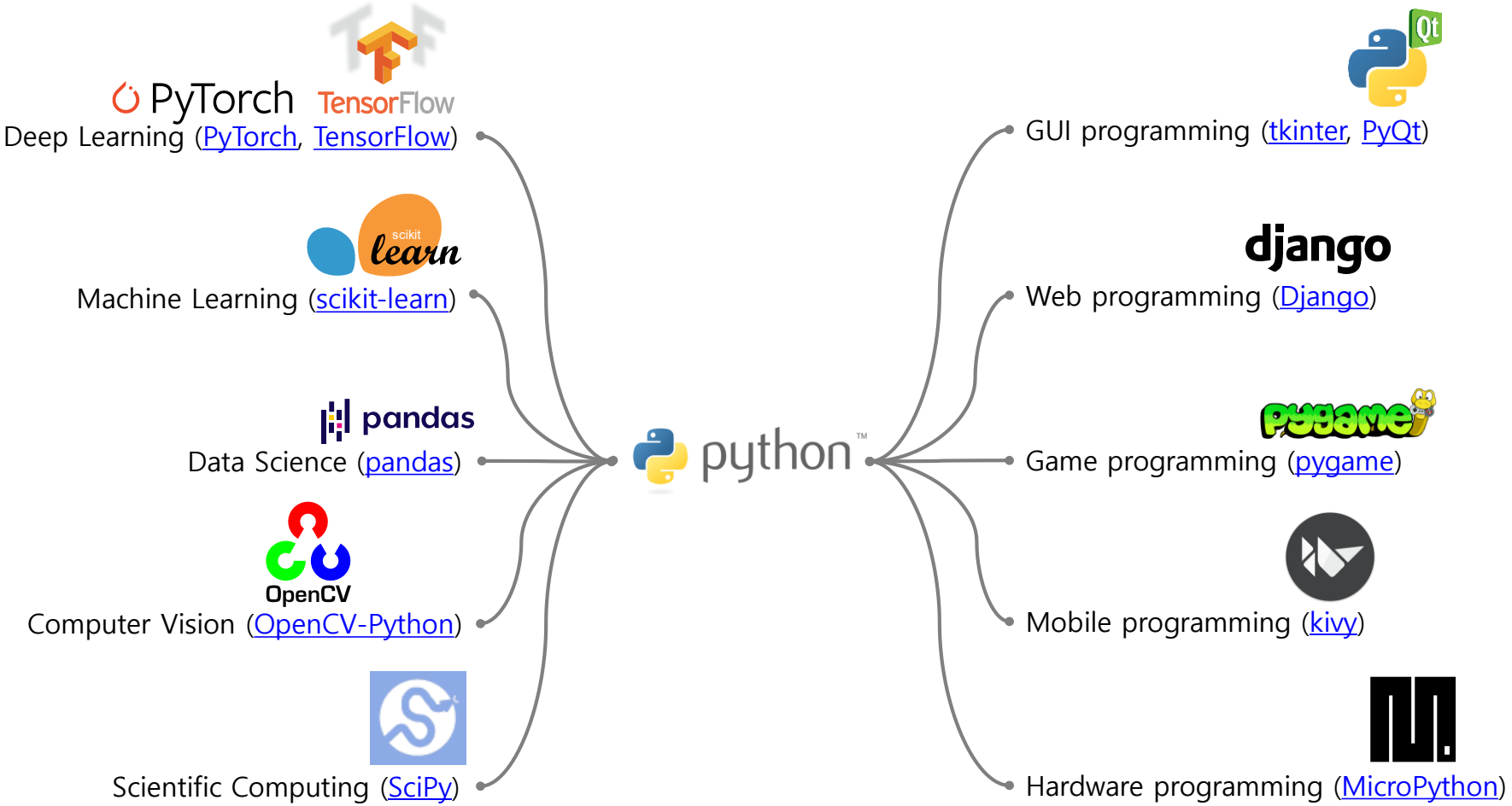
Image: PyPI (retrieved on June 24th, 2021)

- PyPI (The Python Package Index)
  - There are 312,069 projects in Python package repository (on June 24th, 2021).
  - The packages can be easily installed by `pip` (the package installer for Python).



Why Python?

- What can I do with Python?
  - Almost everything you want ~~except hardware programming (e.g. firmware) and mobile programming~~
  - Please visit Awesome Python (by maintained by Vinta Chen).

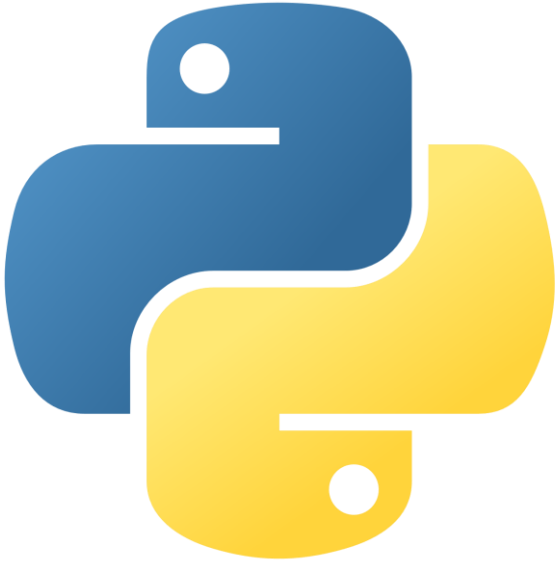


What is Python?

- **Python**
  - A family of snakes
  - A programming language by Guido van Rossum (first released in 1991)
    - Its name is derived from the British comedy group *Monty Python*.

Indian python

Monty Python



Python logo

Image: Wikipedia (Pythonidae, Monty Python)

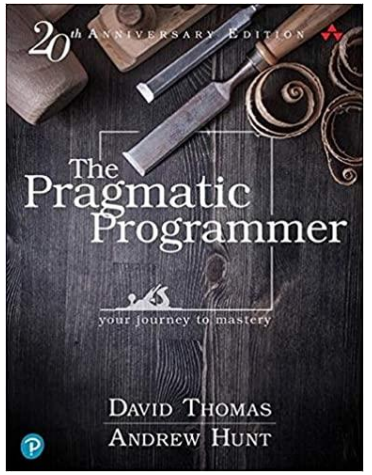
What is Python?

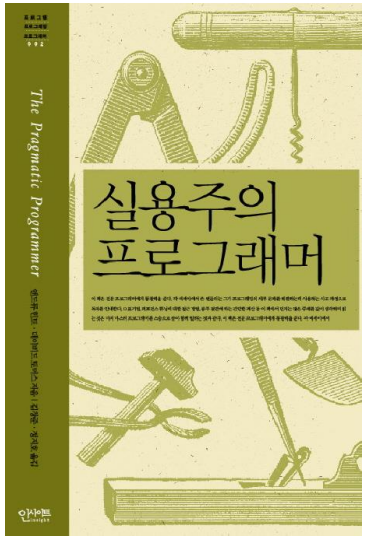
- **Python**
  - Python is an **interpreted**, high-level, general-purpose programming language. [Wikipedia]
    - **Dynamically-typed**, garbage-collected, and **batteries-included**
  - Python is a programming language that lets you work (easily and) quickly and integrate systems more effectively. [Official Homepage]
  - Design philosophy: **Code readability** [PEP 20 – The Zen of Python]
    - e.g. Block range: Curly-bracket `{ ... }` (C/C++, Java) vs. Indentation (Python)
    - e.g. `if` ... `else if` ... `else` ... (C/C++) vs. `if` ... `elif` ... `else` ... (Python)
  - History: v1.0 (1991), v2.0 (2000), v3.0 (2008; *not* completely backward-compatible)
    - cf. C (1972), Objective-C (1984), C++ (1985), R (1993), Java (1995), Java Script (1995), PHP (1995), C# (2000)

- Online references
  - Please visit mint-lab/know-where > Programming > Python.

What is Python?

The Pragmatic Programmer





- The Zen of Python [PEP 20]

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

Image: Amazon and 교보문고

What is Python?

- Compiled language
  - A compiler translates machine code from source code.
  - e.g. C/C++, Java, C#, Objective-C, …
- Interpreted language (cf. script language)
  - An interpreter executes each line of source code step-by-step. (No pre-runtime translation)
  - e.g. **Python**, Java Script, R, PHP, …

- How to use) Python console as a calculator

How to Use) Python on Your Computer

- I am using **Anaconda** (individual license) on Windows.
  - I am using Spyder IDE included in Anaconda. (cf. PyCharm, VS Code, Atom, …)

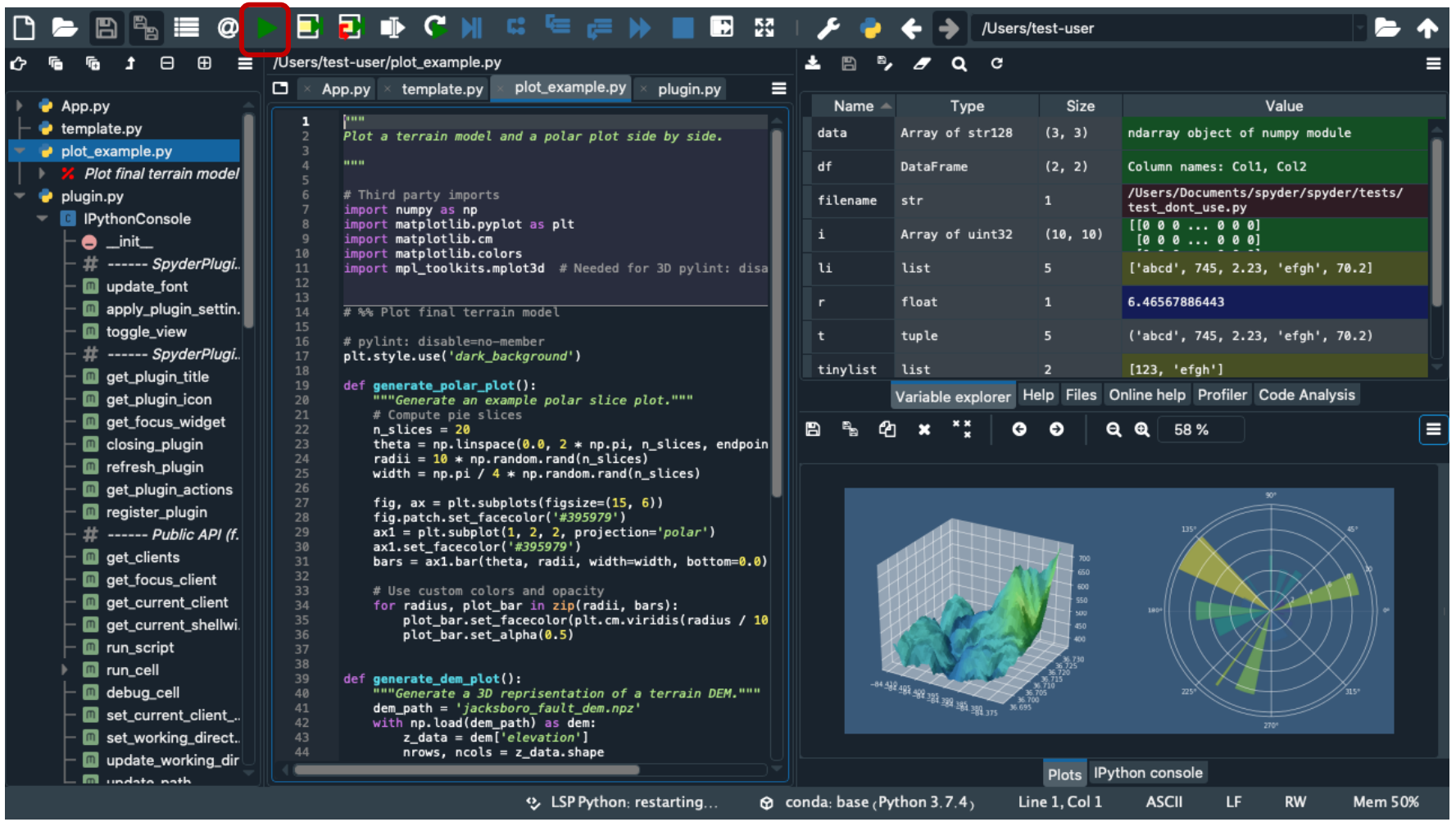


Image: Spyder IDE homepage.

How to Use) Python on Cloud

- I am using **Jupyter Notebook** on **Google Colab**.

  A web-based application that create a document with live code, equations, and visualization

  \+ Zero configuration required
  \+ Free access to GPUs
  \+ Easy sharing

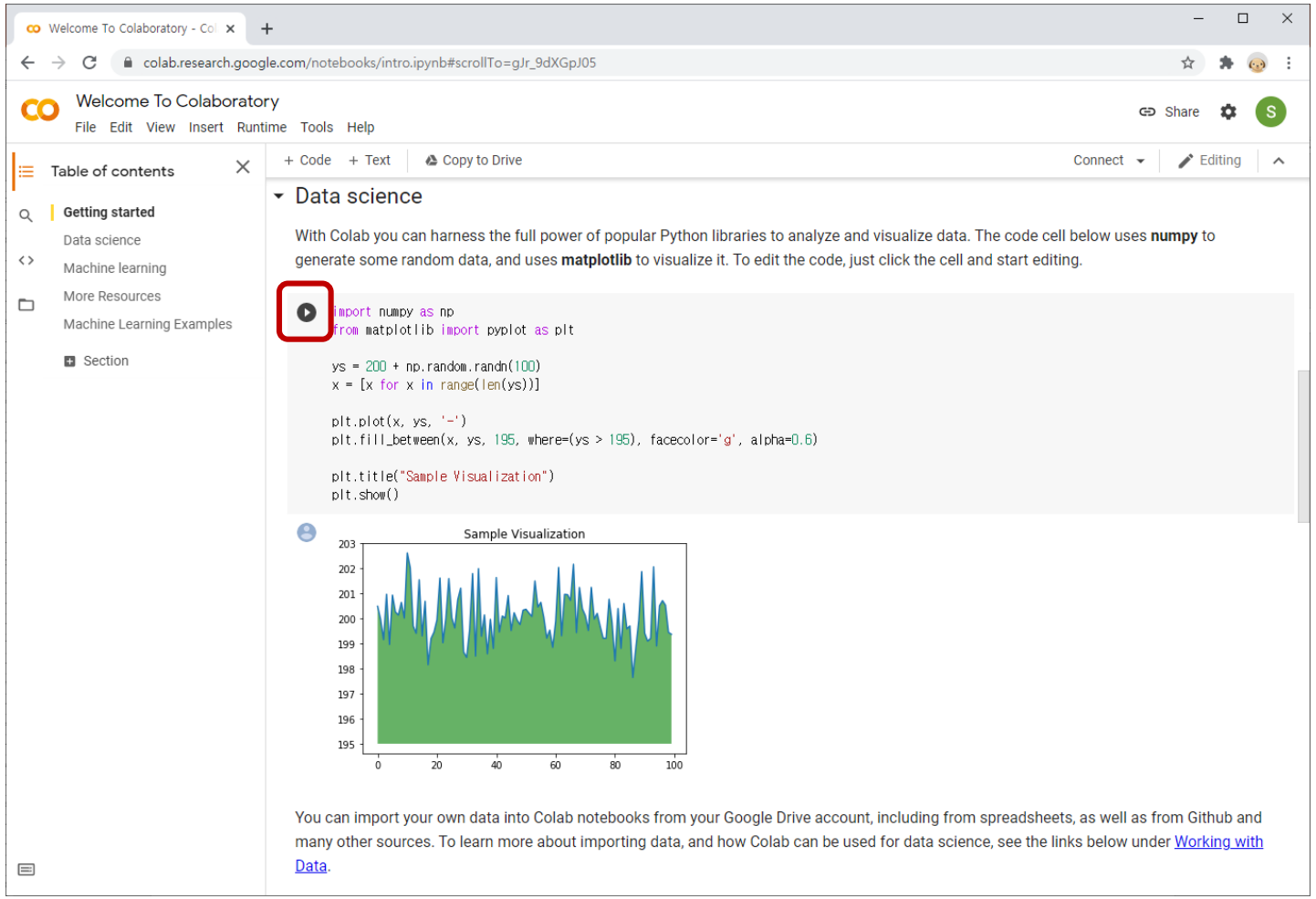
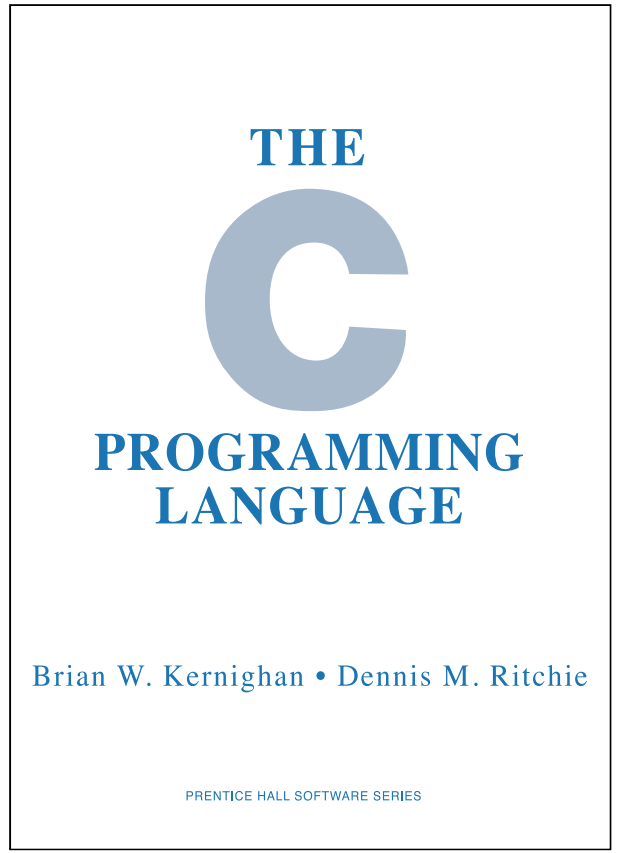


Table of Contents

Hello, World!

```
print('Hello, World!')
print(3.29) # Comment) Not necessary to think about data type
```

Note) `print` is available by default and it accepts any data type (~ `std::cout` in C++).



This lecture assumes that
you already see "Hello, World!"
in other programming languages.

Image: Wikipedia

Data Types

- **Numbers**
  - **`int`**: Integers with an unlimited range

```
a = 3
```

  - **`float`**: Double-precision (64-bit) floating-point numbers

```
a = 3. # Same with 'a = 3.0'
b = 3.29
```

  - **`bool`**: Boolean values (`False` or `True`)

```
a = False
b = (3 == 3.) # Check this result
```

Note) Dynamically-typed → no type specifier

Data Types

- **String**
  - **`str`**: A text encoded in Unicode

```
name = 'Choi'            or   name = "Choi"
```

For multi-line text

```
name = 'Sunglok\nChoi'   or   name = """Sunglok
                              Choi""" # Used for a block comment (~ /* ... */ in C++)
```

Note) String is a built-in type, not in a library (~ `std::string` in C++).

Data Types

- **Compound data**
  - **`tuple`**: Comma-separated arbitrary Python objects

    ```
    prof_tuple = ('Choi', 327, True)  or  prof_tuple = 'Choi', 327, True
    ```

  - **`list`**: A list of arbitrary Python objects (~ `std::array` in C++)

    ```
    prof_list = ['Choi', 327, True]
    ```

  - **`set`**: A unordered set of unique arbitrary objects (~ `std::set` in C++)

    ```
    prof_set =  {'Choi', 327, True}
    prof_set == {'Choi', 327, True, True} # True
    prof_set == {'Choi', True, 327}       # True
    ```

  - **`dict`**: A hash table of arbitrary values indexed by arbitrary keys (~ `std::map` in C++)

    ```
    prof_dict = {'name': 'Choi', 'room_no': 327, 2021: True}
    ```

Note) The *compound* data can contain heterogenous data types (not same with *arrays* with a homogeneous data type).

Data Types

- **Compound data**

Given)
```
prof_str   =  'Choi'
prof_tuple = ('Choi', 327, True)
prof_list  = ['Choi', 327, True]
prof_set   = {'Choi', 327, True}
prof_dict  = {'name': 'Choi', 'room_no': 327, 2021: True}
```

  - **Indexing**

```
prof_tuple[0] == 'Choi'
prof_list[-1] == True # Reverse indexing
prof_set[0]           # Error!
prof_dict['name'] == 'Choi'
prof_dict[2021] == True
```

  - **Slicing**

```
prof_str[1:3] == 'ho'
prof_str[1:] == 'hoi'
prof_str[1::2] == 'hi'
prof_list[::-1] == [True, 327, 'Choi']
```

Data Types

- **Compound data**

  Given)
  ```
  prof_str   =  'Choi'
  prof_tuple = ('Choi', 327, True)
  prof_list  = ['Choi', 327, True]
  prof_set   = {'Choi', 327, True}
  prof_dict  = {'name': 'Choi', 'room_no': 327, 2021: True}
  ```

  – **Concatenation**: Merging two compounds
  ```
  new_str  = prof_str  + ' Sunglok'
  new_list = prof_list + ['Mirae Hall', 'SeoulTech']
  prof_set.union({'Mirae Hall', 'SeoulTech'})
  ```

  – **Appending**: Adding an item to a compound
  ```
  prof_list.append('Mirae Hall')
  prof_set.add('Mirae Hall')
  prof_dict['building'] = 'Mirae Hall'
  ```

  Check) How about concatenation and appending for a **tuple**?

Data Types

- **Useful built-in functions**
  - **Type check**

```
type(prof_str) == str
```

  - **Type casting**

```
int(3.29) == 3
str(3) == '3'
int('29') == 29
```

    Note) The above two conversions are more easy-to-use than C/C++/Java.

  - **Length of compound data**

```
len(prof_name) == 4 # The number of items
```

Operators

- **Operator precedence**

Operator Types	Operators	Description
Compound data (Parentheses)	`(expressions...)`, `[expressions...]`, `{key: value...}`, `{expressions...}`	Binding (tuple) / parenthesized expression, list, dictionary, set
Subscription	`x[index]`, `x[index:index]`, `x(arguments...)`, `x.attribute`	Indexing, slicing, call, attribute reference
Arithmetic (Bitwise)	`**`	Exponentiation
	`+x, -x, ~x`	Positive, negative, bitwise NOT
	`*, @,` `/, //, %`	Multiplication, matrix multiplication, division, floor division, remainder
	`+, -`	Addition and subtraction
Bitwise	`<<, >>`	Bitwise shifts
	`&`	Bitwise AND
	`^`	Bitwise XOR
	`|`	Bitwise OR

Membership Identity Comparison	`in, not in,`
	`is, is not,`
	`<, <=, >, >=, !=, ==`
		Membership tests,
		Identity tests,
		Comparisons

Logical	`not`	Boolean NOT
	`and`	Boolean AND
	`or`	Boolean OR
Ternary	`if – else`	Conditional expression
Lambda	`lambda`	Lambda expression
Assignment	`=, +=, -=, *=, /=`	Assignment expression

Operators

- **Arithmetic** operators

```
type(4 / 2) == float # Always 'float' type (not 'int' type)
(7.5 % 2) == 1.5     # Modulo (remainder)
(7.5 // 2) == 3      # Floor division (integer division; 'int' type)
(-7.5 // 2) == -4
(2 ** 4) == 16       # Exponentiation
```

Note) Please distinguish division (/; `float` type) and floor division (//; `int` type).

- **Logical** operators

```
 not 3.29 > 3  and  10.18 < 10 or 5.12 > 5  # Note) They are not '!', '&&', and '||'.
(not 3.29 > 3) and (10.18 < 10 or 5.12 > 5) # Check two results
```

- **Ternary** operators

```
is_odd = True if x % 2 == 1 else False # In Python
is_odd = (x % 2) == 1 ? 1 : 0;         // In C/C++
```

Flow Control

- **Condition**: `if` statements, ~~`switch` statements~~
- **Loop**: `for` statements, `while` statements
  - Loop control: `break`, `continue`, and `else` statements
- **No action**: `pass` statements (similar to `;` and `{ }` in C/C++)

- Example) Factorial (of a positive integer n)
  - The product of all positive integers less than or equal to n
  - $n! = n \cdot (n-1)! = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 3 \cdot 2 \cdot 1$

```
n = 7      # The given integer
f = 1      # The result of factorial
if n < 0: # Note) No curly-bracket for a block
    pass
elif n == 0:
    pass
else:
    while n > 0:
        f = f * n
        n = n - 1
```

Flow Control

- **Condition**: `if` statements, ~~`switch` statements~~
- **Loop**: `for` statements, `while` statements
  - Loop control: `break`, `continue`, and `else` statements
- **No action**: `pass` statements (similar to `;` and `{ }` in C/C++)

- Example) Prime number
  - A natural number ($n > 1$) that is not a product of two smaller natural numbers

```
n = 7 # The given integer
for x in range(2, n):
    if n % x == 0:
        print(n, 'equals', x, '*', n//x)
        break
else:
    print(n, 'is a prime number')
```

Flow Control

- **Loop**: `for` statements
  - `for` statements with **sequential data** (string, list, tuple, …)

```
year_list = [1982, 1984, 2014, 2016]
for idx in range(len(year_list)):
    print(idx)
for item in year_list:
    print(item)
for idx, item in enumerate(year_list):
    print(idx, item)
```

  - Note) For compound data, you can loop with each index, each item (~ `std::iterator` in C++), and both.

Function Definition

- **Function definition**
  - Example) Factorial (of a positive integer n)
    - The product of all positive integers less than or equal to n
    - $n! = n \cdot (n-1)! = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 3 \cdot 2 \cdot 1$

```
def factorial_for(n):
    f = 1
    for m in range(1, n + 1):
        f *= m
    return f

def factorial_rec(n=1): # Default argument values
    if n <= 0:
        return 1
    else:
        return n * factorial_rec(n - 1)

factorial_for(10) # 3628800
factorial_rec(10) # 3628800
factorial_for()   # Error!
factorial_rec()   # 1
```

Function Definition

- **Multiple return values (as a tuple)**
  - Example) Mean and variance of data
    - Mean (a.k.a. average): $\mu = \mathrm{E}(X)$
    - Variance: $\mathrm{Var}(X) = \mathrm{E}\left((X - \mu)^2\right) = \mathrm{E}(X^2) - \mu^2$

```
def mean_var(data):
    n = len(data)
    if n > 0:
        mean = sum(data) / n
        sum2 = sum([d**2 for d in data])
        var  = sum2 / n - mean**2
        return mean, var
    return None, None

data = [3, 2, 9, 1, 0, 8, 7, 5 ]
pair = mean_var(data)       # pair = (4.375, 9.984)
mean, var = mean_var(data)  # mean = 4.375, var = 9.984
mean, _ = mean_var(data)    # Get only the first one
mean = mean_var(data)[0]    # Get only the first one
```

Object-oriented Programming

- **Class definition** and **object instantiation**
  - Example) Dice and coin

```
from random import randint

class Dice:
    def throw(self):
        return randint(1, 6)

dice = Dice()
print(dice.throw()) # [1, 6]
```

```
from random import randint

class Dice:
    def __init__(self, boundary=(1, 6)): # A constructor
        self.start = min(boundary)
        self.end = max(boundary)

    def throw(self):
        return randint(self.start, self.end)

dice = Dice()
print(dice.throw()) # [1, 6]

coin = Dice((0, 1))
print(coin.throw()) # 0 or 1
```

Summary

- **Why Python? What is Python? How to Use Python**

- **Data Types**
  - Dynamically-typed → no type specifier
  - Built-in compound data types: `str`, `tuple`*, `list`*, `set`*, `dict`* (*support heterogenous data types)
- **Operators**
  - More natural (e.g. `&&` → `and`, `condition ? A : B` → `A if condition else B`)
- **Flow Control**
  - Easier access to elements in compound data with `for` statement
- **Function Definition**
  - Multiple return values (as a `tuple`)
- **Object-oriented Programming**
  - Dynamically-typed → no definition for member variables