

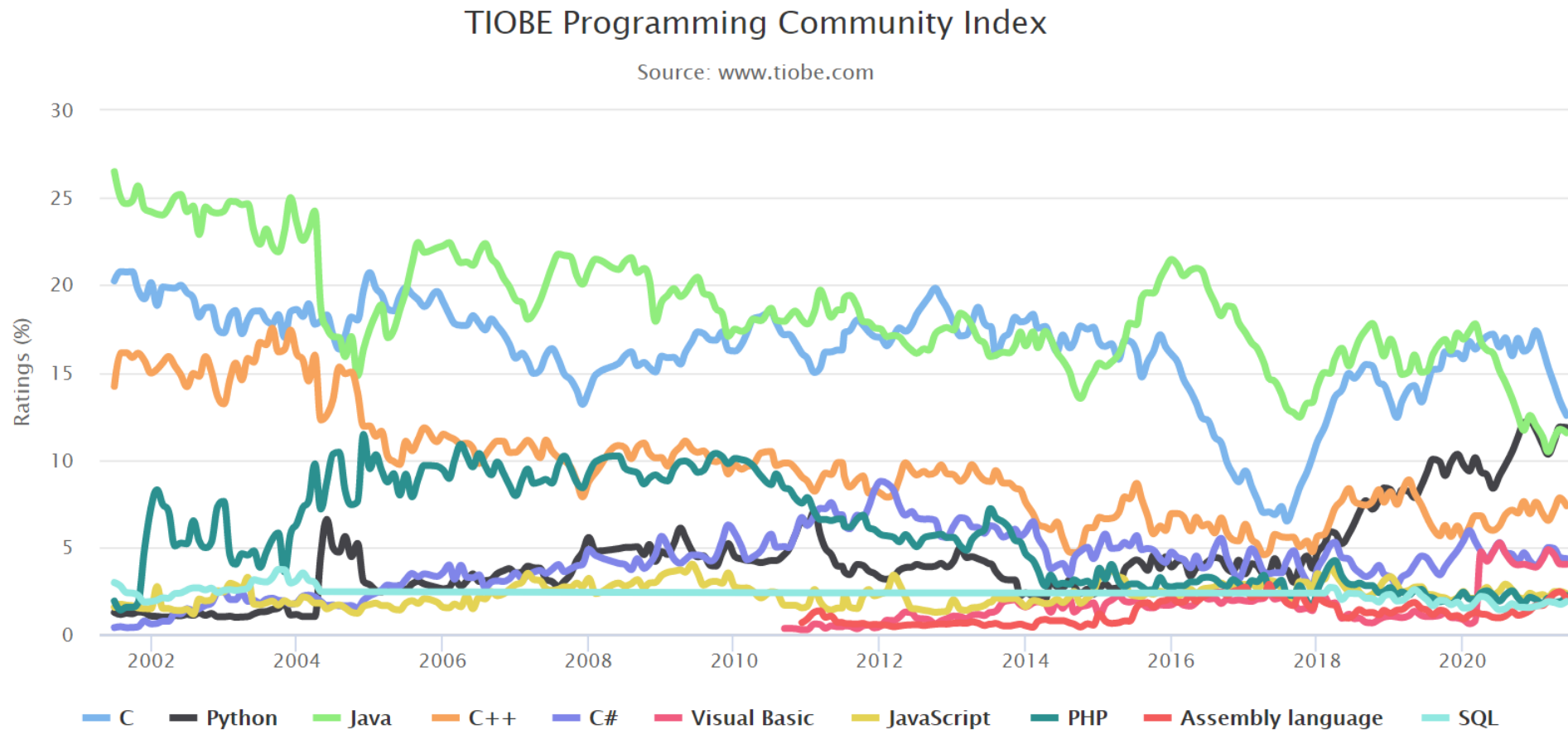
Python: **Basic**

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](https://www.tiobe.com)

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



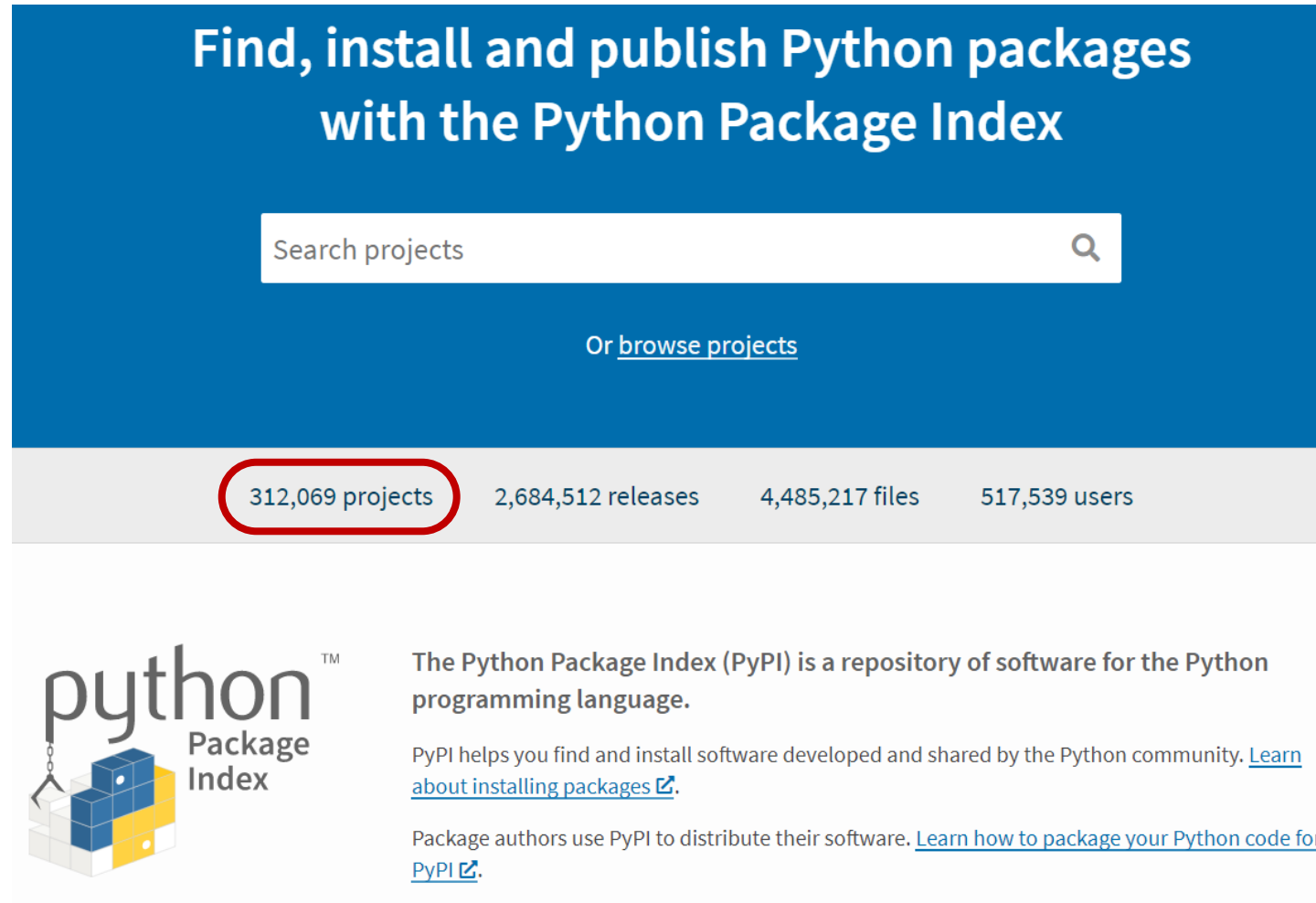
Why Python?

- [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 %

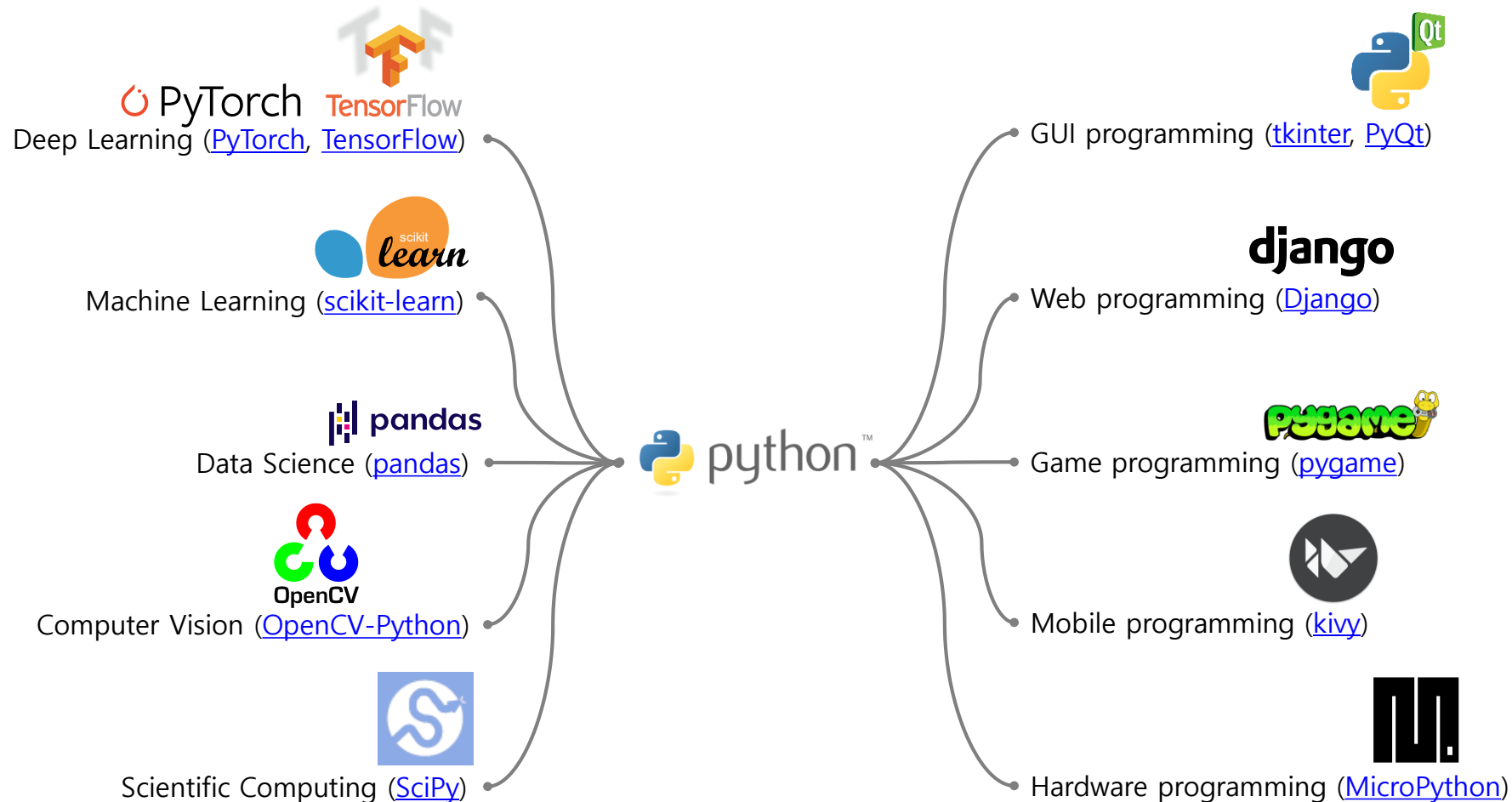
Why Python?

- [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](#) (maintained by Vinta Chen).



What is Python?

- Python

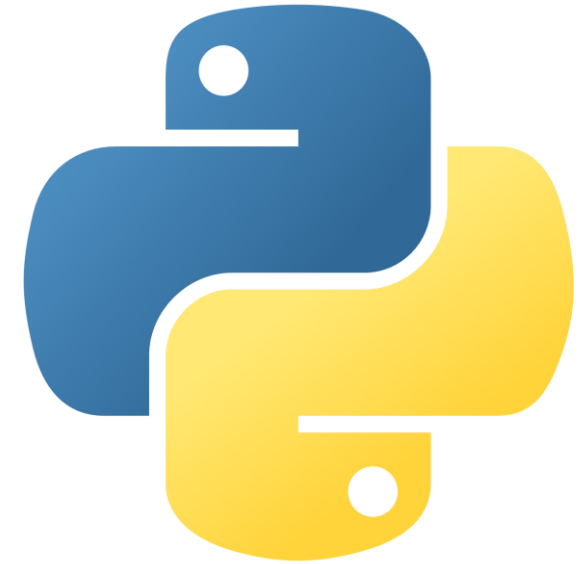
- [A family of snakes](#)
- A programming language by [Guido van Rossum](#) (first released in 1991)
 - Its name is derived from the British comedy group [Monty Python](#).



Indian python



Monty Python



Python logo

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**)
 - Note) C (1972), Objective-C (1984), C++ (1985), R (1993), Java (1995), Java Script (1995), PHP (1995), C# (2000)
- Online references: [MINT Lab's Know-where](#)
 - Please refer *Programming (Python)* category.

What is Python?

- 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.

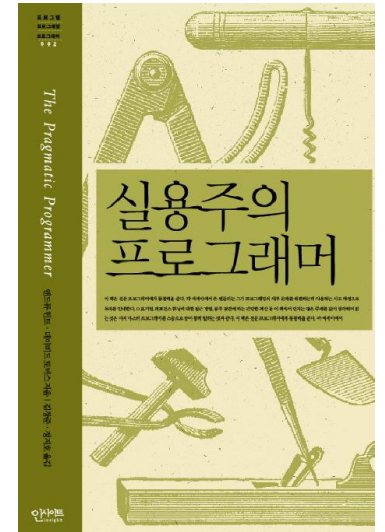
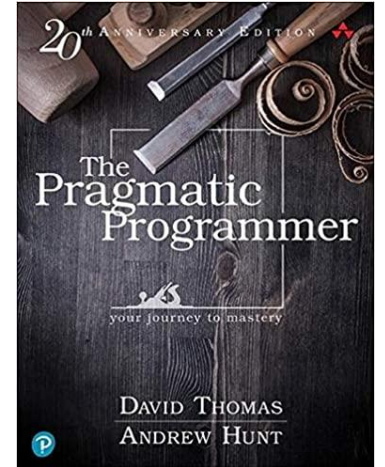
Now is better than never.

Although never is often better than **right** now.

If the implementation is hard to explain, it's a bad idea.

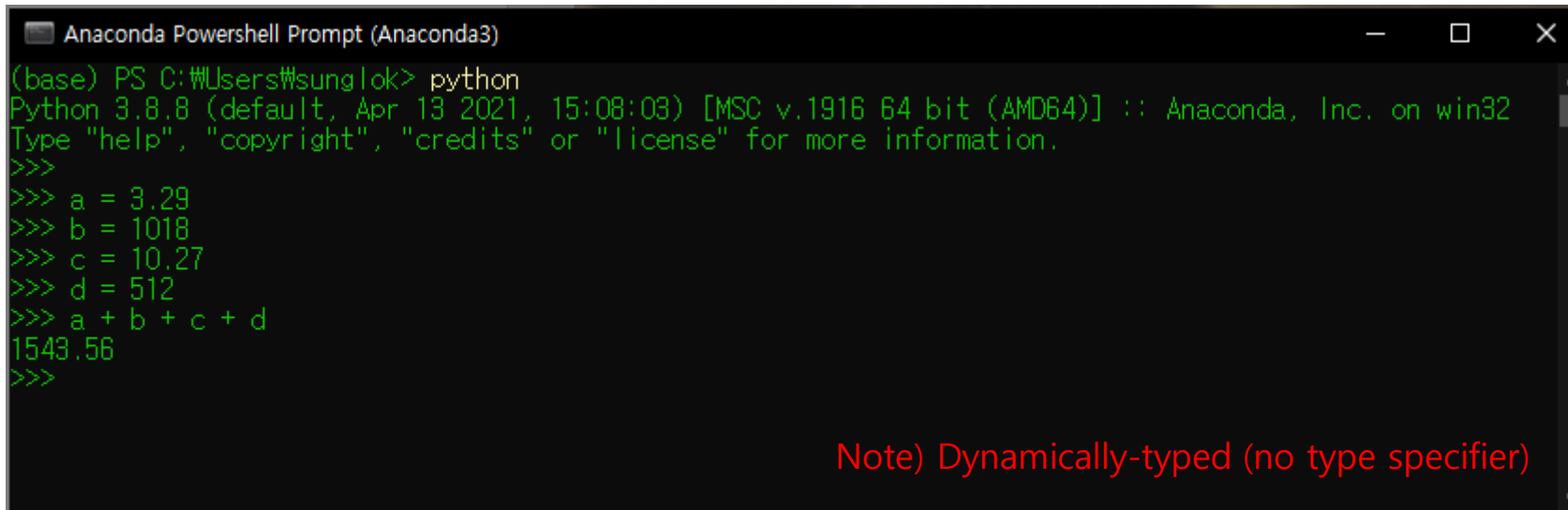
If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!



What is Python?

- [Compiled language](#)
 - A [compiler](#) translates machine code from source code.
 - e.g. C/C++, Java, C#, Objective-C, ...
- Interpreted language (Note: [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

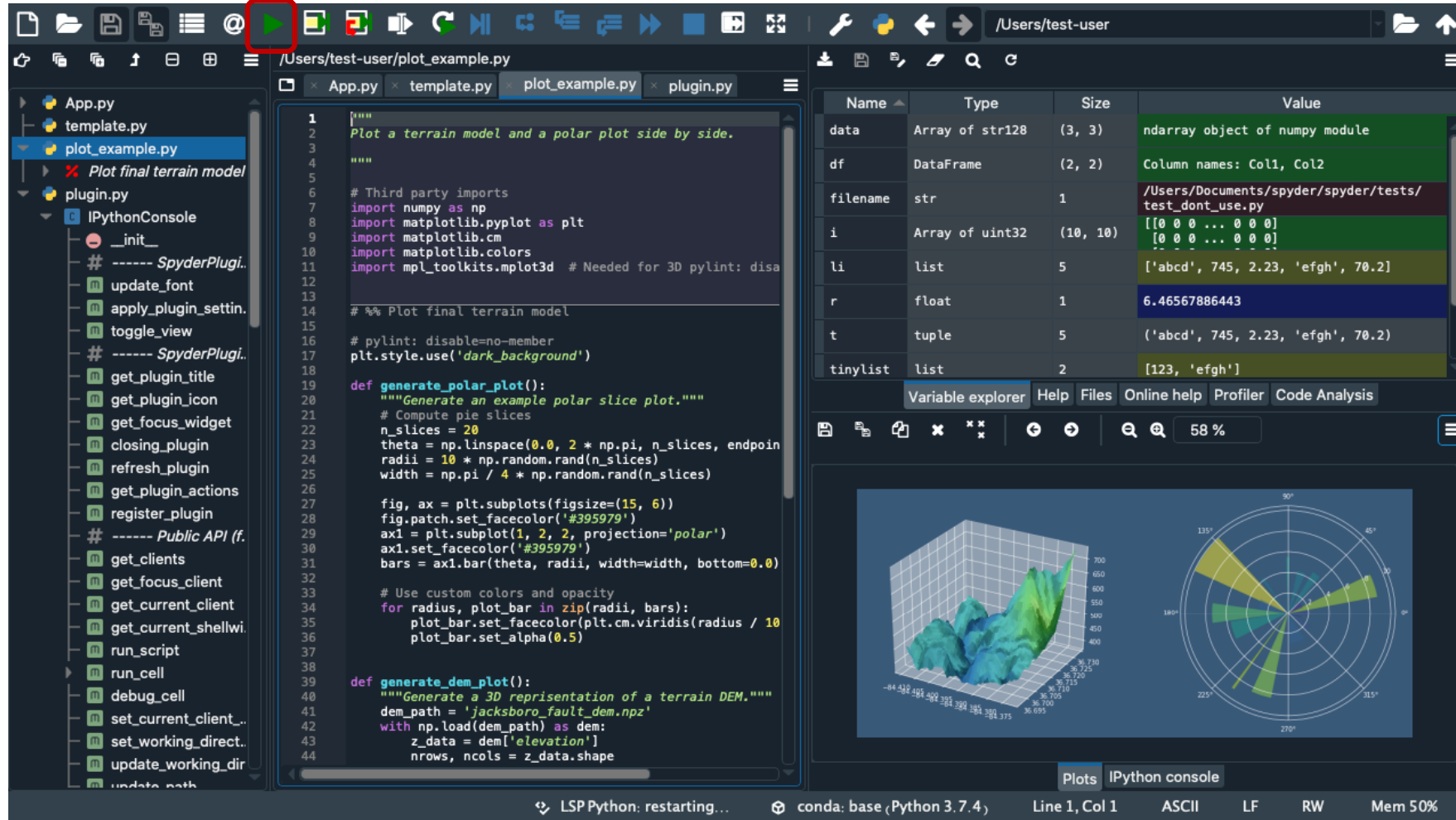


```
(base) PS C:\Users\#sunglok> python
Python 3.8.8 (default, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> a = 3.29
>>> b = 1018
>>> c = 10.27
>>> d = 512
>>> a + b + c + d
1543.56
>>>
```

Note) Dynamically-typed (no type specifier)

How to Use) Python on Your Computer

- I am using [Anaconda](#) (individual license) on Windows.
 - I am using [Spyder IDE](#) included in Anaconda. (Note: [PyCharm](#), [VS Code](#), [Atom](#), ...)



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

Welcome To Colaboratory

File Edit View Insert Runtime Tools Help

Table of contents

- Getting started
- Data science
- Machine learning
- More Resources
- Machine Learning Examples
- Section

+ Code + Text Copy to Drive

Connect Editing

Data science

With Colab you can harness the full power of popular Python libraries to analyze and visualize data. The code cell below uses **numpy** to generate some random data, and uses **matplotlib** to visualize it. To edit the code, just click the cell and start editing.

```
import numpy as np
from matplotlib import pyplot as plt

ys = 200 + np.random.randn(100)
x = [x for x in range(len(ys))]

plt.plot(x, ys, '-')
plt.fill_between(x, ys, 195, where=(ys > 195), facecolor='g', alpha=0.6)

plt.title("Sample Visualization")
plt.show()
```

Sample Visualization

You can import your own data into Colab notebooks from your Google Drive account, including from spreadsheets, as well as from Github and many other sources. To learn more about importing data, and how Colab can be used for data science, see the links below under [Working with Data](#).

Table of Contents

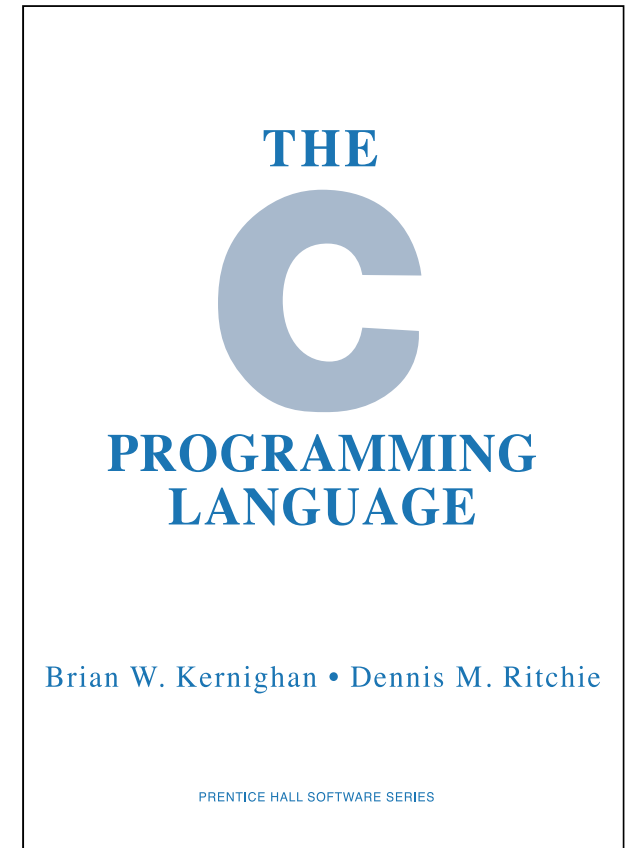
- **Why Python?**
 - Popularity (TIOBE, PYPL)
 - Versatility (*batteries-included*)
- **What is Python?**
 - Interpreted, dynamically-typed, high-level, and general-purpose programming language
 - Design philosophy: Code readability
- **How to Use Python**
- **Data Types**
- **Operators**
- **Flow Control**
- **Function Definition**
- **Object-oriented Programming**

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 meet "Hello, World!"
in other programming languages.

Data Types

▪ Numbers

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

```
a = 329
```

- **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"""` # Also 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', True, 327}          # True (unordered)
```

```
prof_set == {'Choi', 327, True, True} # True (unique)
```

- **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({327, 'Mirae Hall', 'SEOULTECH'})
prof_dict.update({'room_no': 109, 'building': 'Mirae Hall', 'school': '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                                                                   |
|---------------------------------------|--------------------------------------------------------------------------------|-------------------------------------------------------------------------------|
| <b>Compound data (Parentheses)</b>    | (expressions...),<br>[expressions...],<br>{key: value...},<br>{expressions...} | Binding (tuple) / parenthesized expression,<br>list,<br>dictionary,<br>set    |
| <b>Subscription</b>                   | x[index],<br>x[index:index],<br>x(arguments...),<br>x.attribute                | Indexing,<br>slicing,<br>call,<br>attribute reference                         |
| <b>Arithmetic (Bitwise)</b>           | **                                                                             | Exponentiation                                                                |
|                                       | +x, -x, ~x                                                                     | Positive, negative, bitwise NOT                                               |
|                                       | *, @,<br>/, //, %                                                              | Multiplication, matrix multiplication,<br>division, floor division, remainder |
|                                       | +, -                                                                           | Addition and subtraction                                                      |
| <b>Bitwise</b>                        | <<, >>                                                                         | Bitwise shifts                                                                |
|                                       | &                                                                              | Bitwise AND                                                                   |
|                                       | ^                                                                              | Bitwise XOR                                                                   |
|                                       |                                                                                | Bitwise OR                                                                    |
| <b>Membership Identity Comparison</b> | in, not in,<br>is, is not,<br><, <=, >, >=, !=, ==                             | Membership tests,<br>Identity tests,<br>Comparisons                           |
| <b>Logical</b>                        | not                                                                            | Boolean NOT                                                                   |
|                                       | and                                                                            | Boolean AND                                                                   |
|                                       | or                                                                             | Boolean OR                                                                    |
| <b>Ternary</b>                        | if - else                                                                      | Conditional expression                                                        |
| <b>Lambda</b>                         | lambda                                                                         | Lambda expression                                                             |
| <b>Assignment</b>                     | =, +=, -=, *=, /=                                                              | 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

```
x = 3
is_odd = True if x % 2 == 1 else False # In Python: Value1 - Condition - Value2
is_odd = (x % 2) == 1 ? 1 : 0; // In C/C++: Condition - Value1 - Value2
```

# Flow Control

- **Condition:** `if` statement, ~~`switch-case` statement~~ `match-case` statement (Python 3.10+)
- **Loop:** `for` statement, `while` statement
  - Loop control: `break`, `continue`, and `else` statements
- **No action:** `pass` statement (similar to `;` and `{ }` in C/C++)
- Example) Factorial (of a positive integer  $n$ )
  - The product of all positive integers less than or equal to  $n$
  - $n! = n \cdot (n - 1)! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 3 \cdot 2 \cdot 1$

```
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` statement, ~~`switch-case` statement~~ `match-case` statement (Python 3.10+)
- **Loop:** `for` statement, `while` statement
  - Loop control: `break`, `continue`, and `else` statements
- **No action:** `pass` statement (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, ...)
    - You can loop with **each index** or **each item** (~ `std::iterator` in C++) or **both**.

```
year_list = [1982, 1984, 2014, 2016]
for idx in range(len(year_list)): # Loop with indices
 print(idx)
for item in year_list: # Loop with items
 print(item)
for idx, item in enumerate(year_list): # Loop with indices and items
 print(idx, item)
```



# Function Definition

## ▪ Function definition

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

```
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):  $\mu = E(X) = \frac{1}{n}(x_1 + x_2 + \dots + x_n)$
    - Variance:  $\text{Var}(X) = E((X - \mu)^2) = E(X^2) - \mu^2$

```
def mean_var(data):
 n = len(data)
 if n > 0:
 mean = sum(data) / n
 sum2 = 0
 for datum in data:
 sum2 += datum**2
 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