INF1002
Programming Fundamentals
Lecture 1: Python basic I

Zhang Zhengchen
zhengchen.zhang@singaporetech.edu.sg

Outline

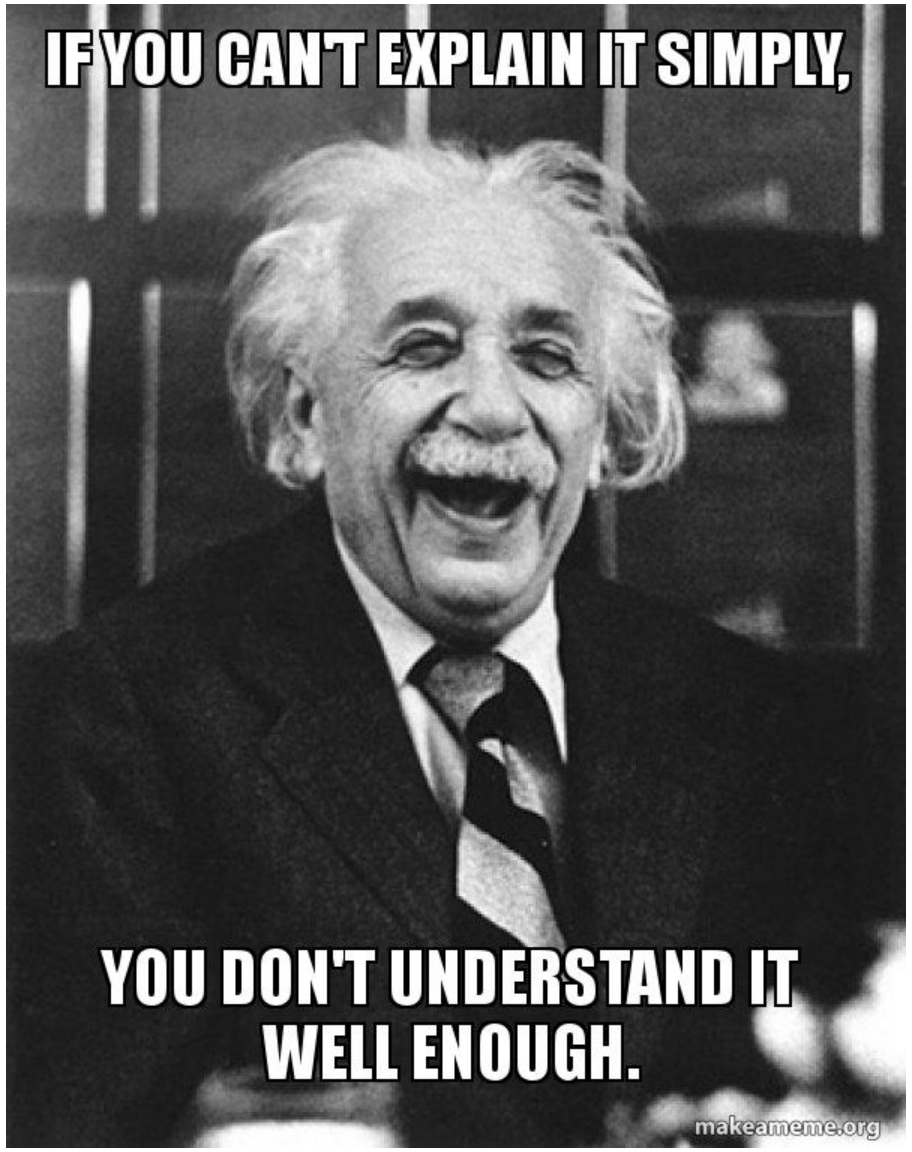
- Module introduction
- **Python introduction**
- Some concept of software engineering
- Python basic I
- Some concept of programming

Python

First Steps in Python

Why Python?

- Simple
- Powerful
  - Community
  - Ecosystem
  - Application in Machine Learning



TIOBE
the software quality company

About us | Knowledge | News | Coding Standards | TIOBE Index | Contact

Products | Quality Models | Markets | Schedule a demo

The index can be used to check whether your programming skills are still up to date or to make a strategic decision about what programming language should be adopted when starting to build a new software system. The definition of the TIOBE index can be found here.

Jun 2024	Jun 2023	Change	Programming Language	Ratings	Change
1	1		Python	15.39%	+2.93%
2	3	↑	C++	10.03%	-1.33%
3	2	↓	C	9.23%	-3.14%
4	4		Java	8.40%	-2.88%
5	5		C#	6.65%	-0.06%
6	7	↑	JavaScript	3.32%	+0.51%
7	14	⇈	Go	1.93%	+0.93%
8	9	↑	SQL	1.75%	+0.28%
9	6	↓	Visual Basic	1.66%	-1.67%
10	15	⇈	Fortran	1.53%	+0.53%
11	11		Delphi/Object Pascal	1.52%	+0.27%
12	19	⇈	Swift	1.27%	+0.33%
13	10	↓	Assembly language	1.26%	-0.03%

TIOBE Programming Community Index: an index of evaluating language popularity
https://www.tiobe.com/tiobe-index/

Why is Python So Popular?

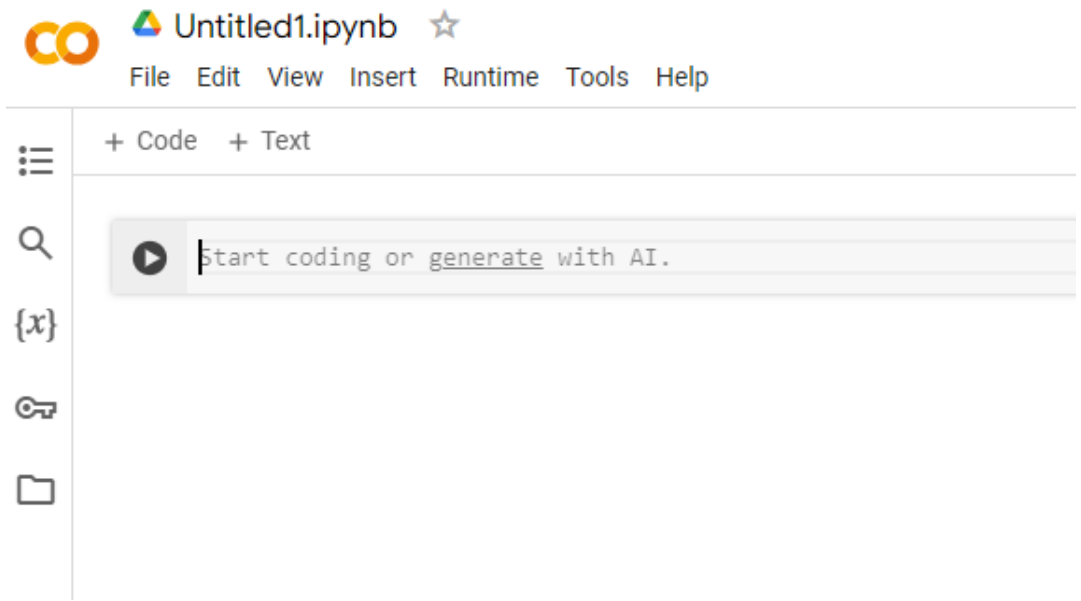
- Simple
- Powerful
  - Ecosystem
  - Community
  - Cross-platform
  - Application in Machine Learning

The Zen of Python

- 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!

Your First Python Program

- https://colab.research.google.com/
  - File → New notebook in Drive
  - Rename, Save, …



Your First Python Program

- Write one program to print "Hello World!" in Python
  - print ("Hello World!")
  - Or
  - print ('Hello World!')
- print()
  - to print something to the screen
  - Using either " " or ' ' around what you want to print
  - Further Reading
    - The arguments of print() function

Knowing Python

- Single Quote and Double Quote can generally be used interchangeably.
  - print('Hello, World! ')
  - print("Hello, World! ")
  - print('He said, "Python is awesome! "')
  - print("It's a beautiful day! ")

Knowing Python

- What if I want to print
  - It's a "beautiful" day.

- Use a backslash ('\') to escape quotes inside the string

```
print('It\'s a "beautiful" day! ')
print("It's a \"beautiful\" day! ")
```

- Escape characters

\n	Newline
\t	Tab
\\	backslash

Knowing Python

- Raw strings
  - Prefixing a string with r or R makes it a raw string, where backslashes are not treated as escape characters

  - print(r'it\n1\t2')
  - print('it\n1\t2')

  - Print(r"C:\Users\Name")
  - Print("C:\Users\Name")

Knowing Python

- The language is case sensitive
  - print ≠ Print ≠ PRINT

- Indention matters
  - Some development tools would make automatic indention
  - Be careful about indention
    - In nested if, for, while

Write comments

- Comments
  - People may read your code
  - Guide your thinking
  - In a single line
    - Using the symbol #
  - For multiple lines
    - Using three quotation marks ''' '''
    - Docstring

```
# Open the first file
f_first = open('first.txt','r')
# Store data in First_Data

# Open the second file

# Append the data in First_Data to the second file
```

Write comments

- Comments
  - People may read your code
  - Guide your thinking
  - In a single line
    - Using the symbol #
  - For multiple lines
    - Using three quotation marks ''' '''
    - Docstring

```
# Open the first file
f_first = open('first.txt','r')
# Store data in First_Data

# Open the second file

# Append the data in First_Data to the second file
```

Python 2 and Python 3

- We use Python 3 in this module
- Print statement
  - Python 2: print "abc ", print 'abc'
  - Python 3: print("abc")
- Libraries
  - Some libraries only work on P2 e.g. PyImage
  - P3 is increasing the third party module support (especially those machine/deep learning libraries)
- Others
  - **Unicode and Strings**
  - Division with integers
  - Input() is now safe to use
- More difference can be found at: https://www.interviewbit.com/blog/difference-between-python-2-and-3/

Review

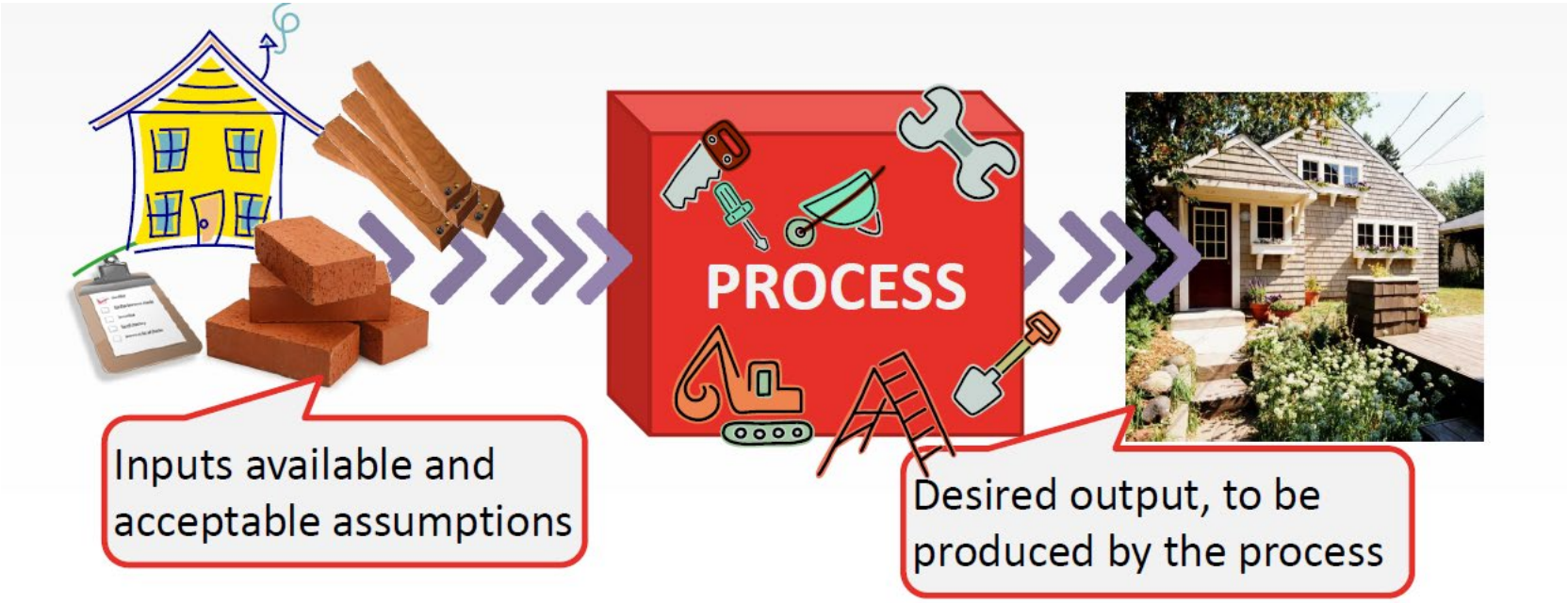
- Hello World in python
- Quotation marks
- Escape Characters
  - Backslash
- Comments
- Versions of Python

Outline

- Module introduction
- Python Introduction
- **Some concept of software engineering**
- Python basic I
- Some concept of programming

Programming and Problem Solving

- A problem is defined by its inputs and the desired property of the output.



Evans D. Introduction to computing: explorations in language, logic, and machines[M]. CreateSpace Independent Publishing Platform, 2011.

How to solve a problem

HOW TO SOLVE IT

UNDERSTANDING THE PROBLEM

First.

You have to *understand* the problem.

What is the unknown? What are the data? What is the condition? Is it possible to satisfy the condition? Is the condition sufficient to determine the unknown? Or is it insufficient? Or redundant? Or contradictory?

Draw a figure. Introduce suitable notation.

Separate the various parts of the condition. Can you write them down?

DEVISING A PLAN

Second.

Find the connection between the data and the unknown. You may be obliged to consider auxiliary problems if an immediate connection cannot be found. You should obtain eventually a *plan* of the solution.

Have you seen it before? Or have you seen the same problem in a slightly different form?

Do you know a related problem? Do you know a theorem that could be useful?

Look at the unknown! And try to think of a familiar problem having the same or a similar unknown.

Here is a problem related to yours and solved before. Could you use it? Could you use its result? Could you use its method? Should you introduce some auxiliary element in order to make its use possible?

Could you restate the problem? Could you restate it still differently? Go back to definitions.

Pólya G. How to solve it: A new aspect of mathematical method[M]. Princeton university press, 2014, first published in 1945.

How to solve a problem

If you cannot solve the proposed problem try to solve first some related problem. Could you imagine a more accessible related problem? A more general problem? A more special problem? An analogous problem? Could you solve a part of the problem? Keep only a part of the condition, drop the other part; how far is the unknown then determined, how can it vary? Could you derive something useful from the data? Could you think of other data appropriate to determine the unknown? Could you change the unknown or the data, or both if necessary, so that the new unknown and the new data are nearer to each other?

Did you use all the data? Did you use the whole condition? Have you taken into account all essential notions involved in the problem?

CARRYING OUT THE PLAN

Third.
Carry out your plan.

Carrying out your plan of the solution, *check each step*. Can you see clearly that the step is correct? Can you prove that it is correct?

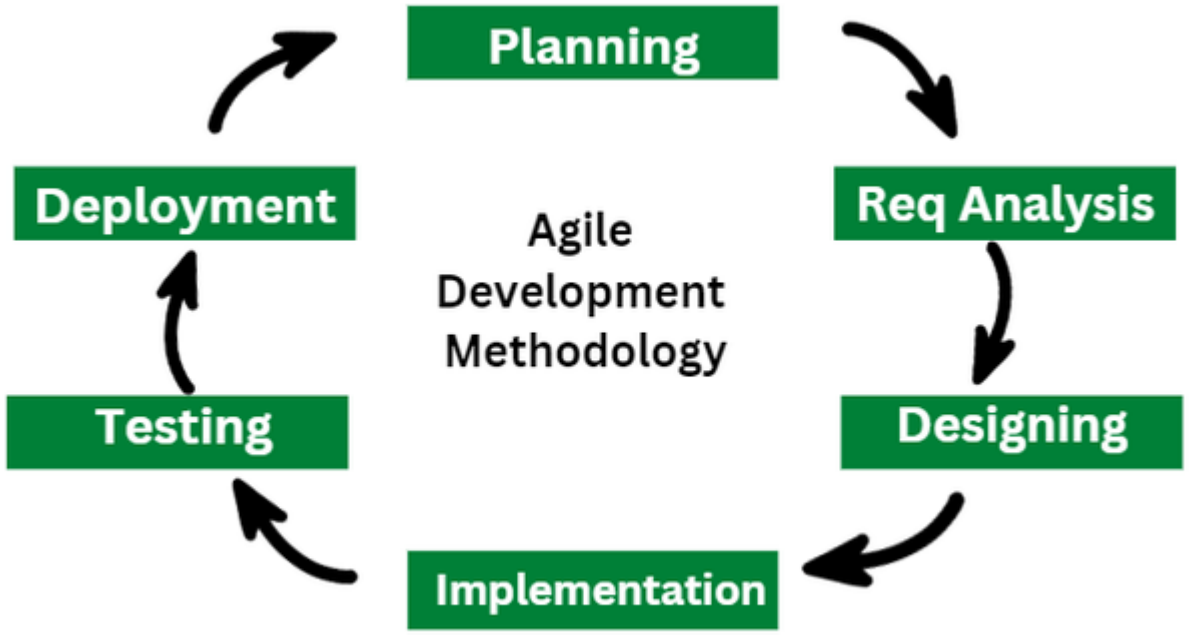
LOOKING BACK

Fourth.
Examine the solution obtained.

Can you *check the result?* Can you check the argument?
Can you derive the result differently? Can you see it at a glance?
Can you use the result, or the method, for some other problem?

Software Engineering

- Software Development Methodologies



Agile Development in Software Engineering

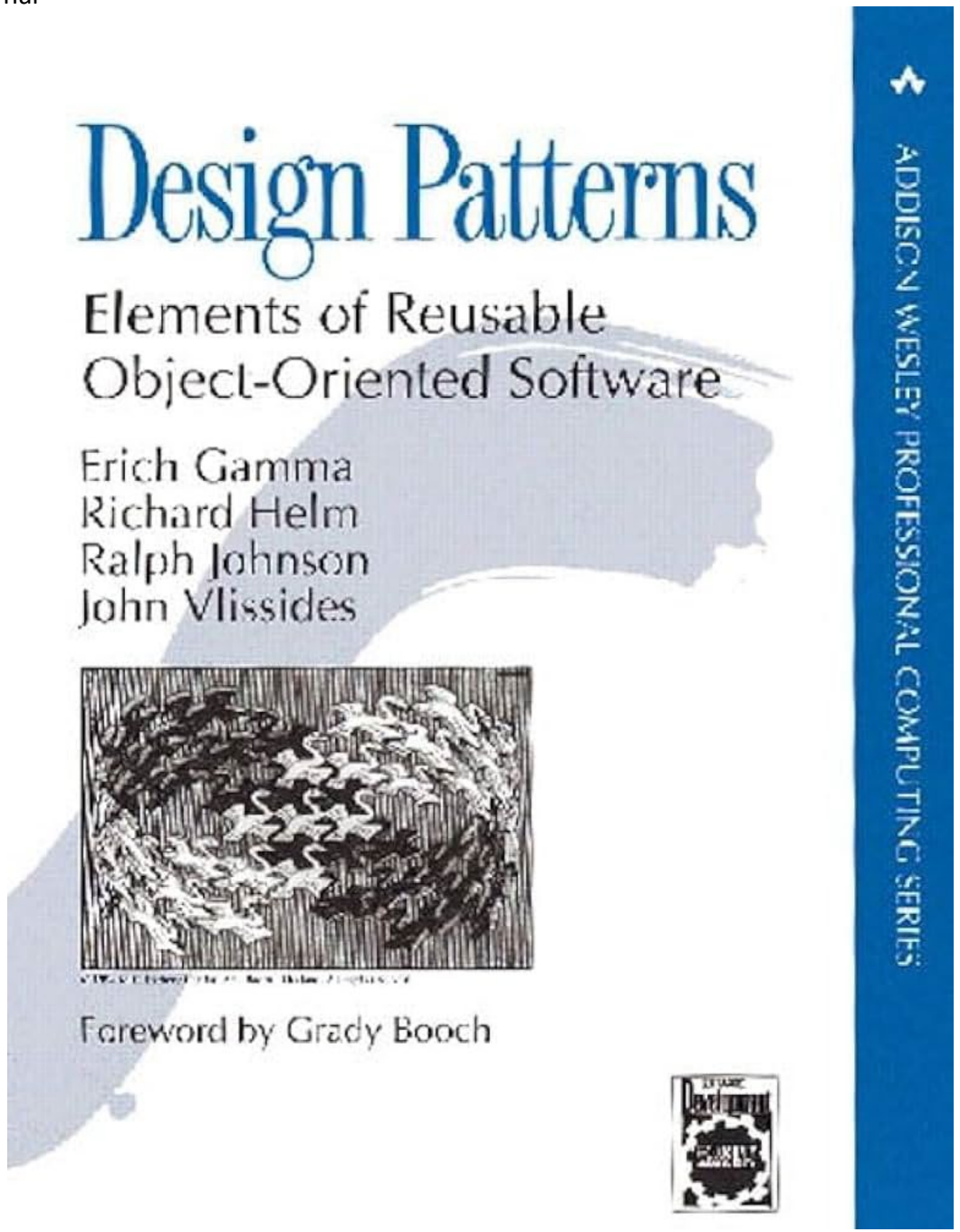
https://www.geeksforgeeks.org/what-is-agile-methodology/

Know the problem first

- Requirement Analysis
  - Let's store files into a database
    - What kind of files
    - File size, numbers
    - Will modify or not
    - Will read them frequently
    - ...

Design

- Design Pattern
  - Reusable solutions in software development that addresses common design problems, promoting code **maintainability**, **scalability**, and **reusability**.

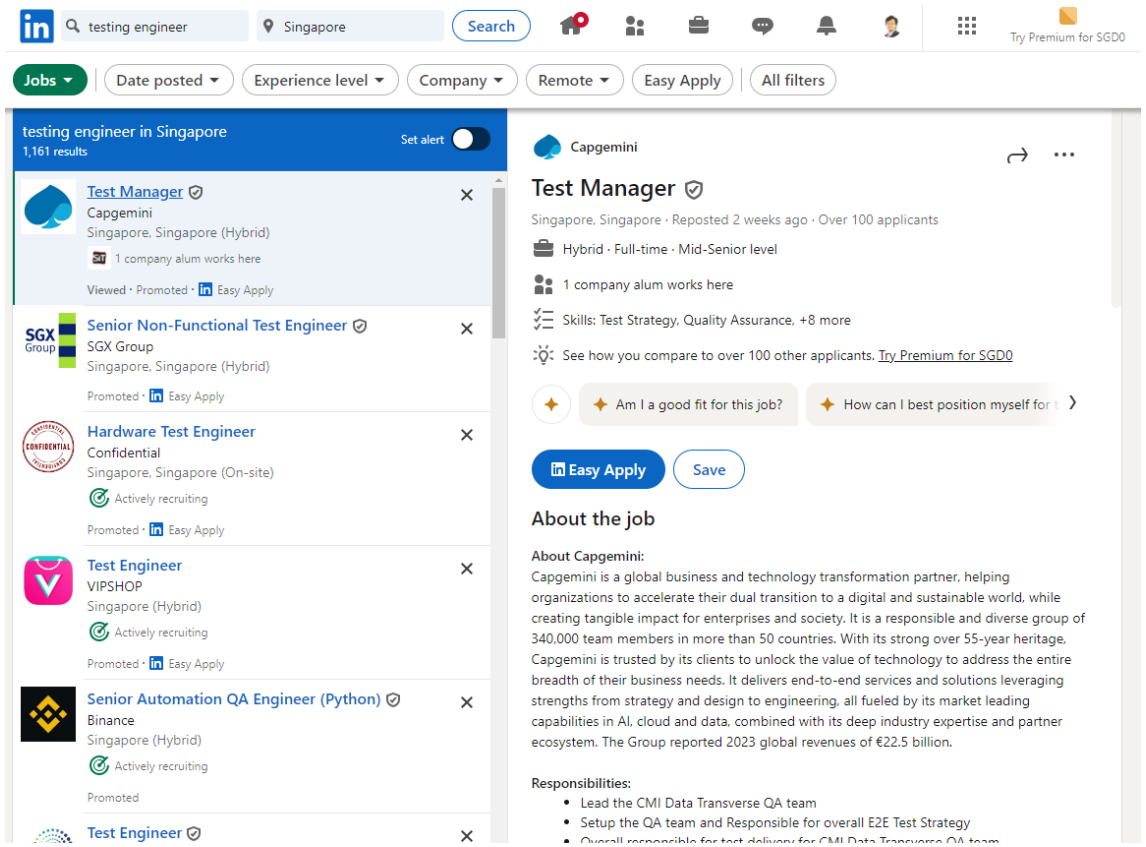


Implementing - Debug

- Logic Error
  - When the code is executed, it produces incorrect results
  - Known as a bug
- Visual Studio Code
  - Run: F5
  - Breakpoint: pause the execution of a program at a specific line of code
  - Step over: F10
  - Step into: F11

Testing

- The act of checking whether software satisfies expectations.
- Automated testing
- Levels
  - Unit testing
  - Integration testing
  - System testing
- Test Cases
- Make your program robust



Deployment

- Take a website as an example
  - Develop on Local Machine
    - You create the website and test it on your laptop.
  - Test on Staging Server
    - You upload it to a staging server to simulate the production environment and catch any bugs.
  - Deploy to Production Server
    - Once everything works perfectly, you upload it to a production server.
    - Now, anyone can visit your website using its URL.
  - Monitor
    - After deployment, you regularly check the website to ensure it’s running smoothly and fix any issues visitors report.

Deployment

- Key Concepts:
  - Continuous Integration/Continuous Deployment (CI/CD)
    - A set of practices that involve automatically testing and deploying code changes to ensure rapid and reliable updates.
  - Version Control
    - Tools like Git help manage changes to the code and keep track of different versions.
  - Rollback
    - The ability to revert to a previous version if the new deployment causes issues.
  - Large scale and complex real world problems

Looking back

- Monitor your products
- Get feedback
- Evaluate your methods/solutions
  - With numbers
- Evaluation
  - A formal end of this step
  - A guide to your next step

Review

- Programming and problem solving
  - Understand the problem
  - Plan
  - Execute the plan
  - Lookback
- Programming and software engineering
  - Plan
  - Requirement Analysis
  - Design
  - Implementation
  - Testing
  - Deployment
  - Maintenance
  - Iteration…

Outline

- Module introduction
- Python Introduction
- Some concept of software engineering
- **Python basic I**
- Some concept of programming

Variables and Assignment Statements

Module Road Map

Variables & Types & Operators

Selection & Iteration

Advanced Data Structures

Functional abstraction

Recursion

Higher order function

Real World Problems / Project Management Problems/ Innovation / Self-study

Variables and Assignment Statements

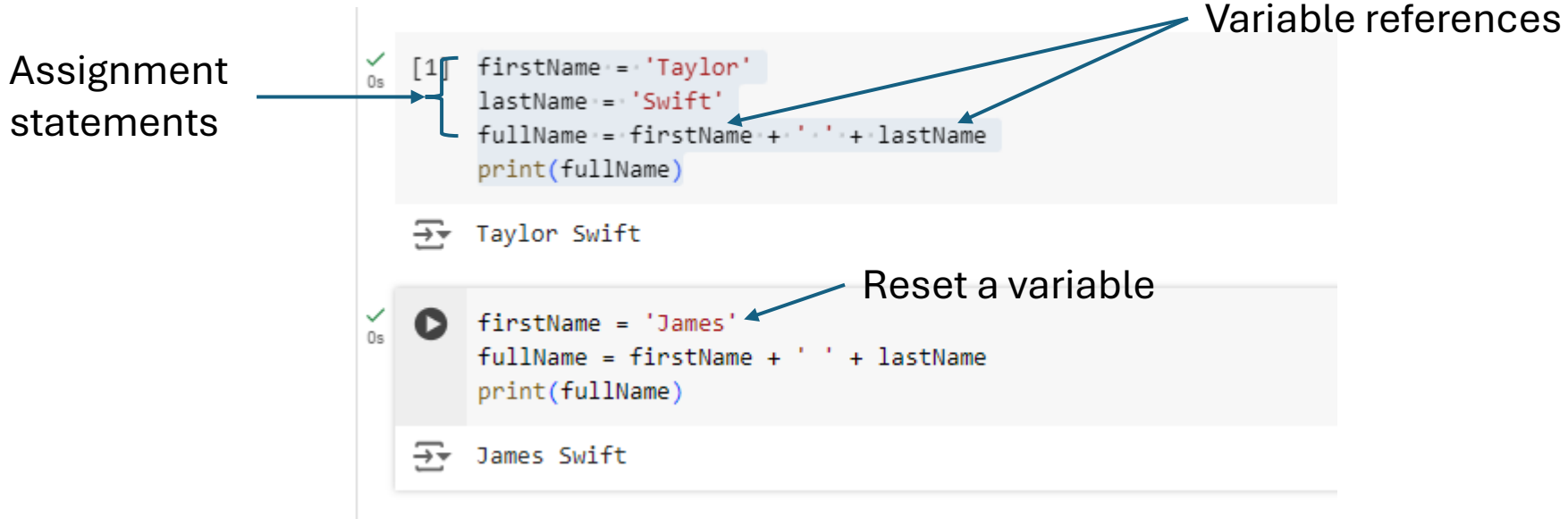
- Variables
  - <variable name > = expression
    - print('Hello, World! ')
    - print("Hello, World! ")
    - print('He said, "Python is awesome! "')
    - print("It's a beautiful day! ")
  - print('Taylor')
  - firstName = 'Taylor'
  - print(firstName)

Variables and Assignment Statements

- Variables
  - <variable name > = expression
  - A variable is a **reserved memory location** used to **store values**

Variables and Assignment Statements

- Variables
  - <variable name > = expression
  - A name
    - We can use the name in following expressions
    - **Variable references**
    - Values can be reset to new ones
    - **Assignment statement**



Variable Naming Rules

- Variables must **start with a letter or an underscore _**
- Subsequent characters can be **letters, numbers, or underscores**
- Variable names are **case-sensitive**
- Keywords **cannot** be used as variable names:
  - Examples of keywords: **if, else, for, while, class, def, return, try, except**, etc.

- Which one is valid?
  - _myvar, Var1, 1var, -var, return, good

Variable Naming Rules

- Variables must **start with a letter or an underscore _**
- Subsequent characters can be **letters, numbers, or underscores**
- Variable names are **case-sensitive**
- Keywords **cannot** be used as variable names:
  - Examples of keywords: **if, else, for, while, class, def, return, try, except**, etc.

- Which one is valid?
  - _myvar, Var1, 1var, -var, return, good

Variable Naming Rules

- Use lowercase letters and underscores for variable names (snake_case):
  - Example: my_variable, user_name, total_sum
- Constants (values that do not change) are usually written in all uppercase with underscores (UPPERCASE_WITH_UNDERSCORES):
  - Example: PI = 3.14159, MAX_LIMIT = 100
- Use meaningful variable names:
  - Choose names that clearly describe the variable's purpose, e.g., total_sum instead of ts.
  - Avoid single-character variable names, unless in loops (e.g., i, j, k):

Variable Naming Rules

- Follow the PEP 8 style guide:
  - PEP 8 is the Python community's style guide, recommending consistent and readable naming practices.
  - https://peps.python.org/pep-0008/

- Python 3 supports Unicode characters, allowing variable names in non-Latin scripts like Chinese or Japanese, but it's best to maintain consistency and readability.

Variables and Assignment Statements

- Variables
  - <variable name > = expression
  - A variable is a **reserved memory location** used to **store values**
  - How can I know the memory address of a variable?
  - id()

```
print('Taylor')
firstName = 'Taylor'
print(firstName)
print(id(firstName))
```

```
Taylor
Taylor
139087464109552
```

Multiple Assignment

```
a=b=c=10
print(a,b,c)
print(id(a),id(b),id(c))
```

```
10 10 10
135632150462992 135632150462992 135632150462992
```

```
[25] a,b,c=10,20.0,'third'
     print(a,b,c)
```

```
10 20.0 third
```

```
a,b,c=10
print(a,b,c)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-26-88b536826383> in <cell line: 1>()
----> 1 a,b,c=10
      2 print(a,b,c)

TypeError: cannot unpack non-iterable int object
```

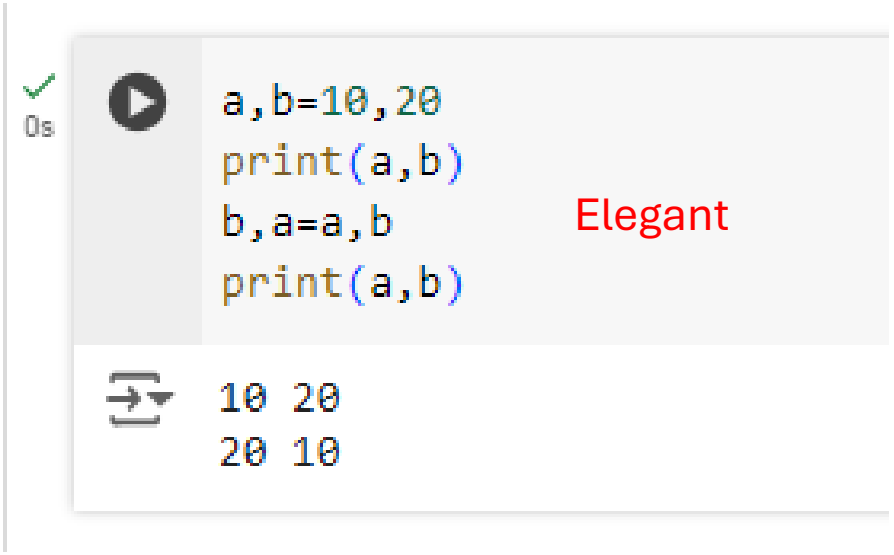
Next steps: Explain error

Multiple Assignment

- Practice
  - How can we swap the values of two variables?
  - a = 10, b = 20
  - a = 20, b = 10

Multiple Assignment

- Practice
  - How can we swap the values of two variables?
  - a = 10, b = 20
  - a = 20, b = 10



Variable types

- Variables
  - Name
  - Id
  - Type

- How to know a variable's type?
- type()

Variable types

Type	Samples
int	8,12,1024
float	2.3, 3.1415926
bool	True, False
str	'Hello, World! ', '3.1415926'
None	None
List	
Tuple	
Set	
Dictionary	
byte	

Type Casting and Dynamic Typing

- Type casting

```
[2]  x = 10
     print(x,type(x))
     y = str(x)
     print(y,type(y))
```

```
10 <class 'int'>
10 <class 'str'>
```

- Dynamic Typing

```
x = 10
print(x,type(x))
x = 'string'
print(x,type(x))
```

```
10 <class 'int'>
string <class 'str'>
```

Type Casting

- **Numbers to string**
- String to numbers
- Numeric precision adjustment

```
salary = 100
print(type(salary))
tax = salary * 0.7
print(type(tax))
print('tax is '+tax)
```

```
<class 'int'>
<class 'float'>
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-b59593431947> in <cell line: 5>()
      3 tax = salary * 0.7
      4 print(type(tax))
----> 5 print('tax is '+tax)

TypeError: can only concatenate str (not "float") to str
```

Next steps: Explain error

```
salary = 100
print(type(salary))
tax = salary * 0.7
print(type(tax))
print('tax is '+str(tax))
```

```
<class 'int'>
<class 'float'>
tax is 70.0
```

Type Casting

- Numbers to string
- **String to numbers**
- Numeric precision adjustment

```
salary = input('Please input your salay: ')
print(salary,type(salary))
tax = salary * 0.2
print(tax)
```

```
Please input your salay: 1000
1000 <class 'str'>
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-103fcfa27bf2> in <cell line: 3>()
      1 salary = input('Please input your salay: ')
      2 print(salary,type(salary))
----> 3 tax = salary * 0.2
      4 print(tax)

TypeError: can't multiply sequence by non-int of type 'float'
```

Next steps: Explain error

```
salary = input('Please input your salay: ')
print(salary,type(salary))
salary = float(salary)
print(salary,type(salary))
tax = salary * 0.2
print(tax)
```

```
Please input your salay: 1000
1000 <class 'str'>
1000.0 <class 'float'>
200.0
```

Type Casting

- Numbers to string
- String to numbers
- **Numeric precision adjustment**
  - **Int and float**

```
tax = 70.9
int_tax = int(tax)
print(int_tax)
print(type(int_tax))
str_tax = str(tax)
print(str_tax)
print(type(str_tax))
```

```
70
<class 'int'>
70.9
<class 'str'>
```

Why 70, not 71?

```
salary = -100.9
int_salary = int(salary)
print(int_salary)
print(type(int_salary))
round_salary = round(salary)
print(round_salary)
print(type(round_salary))
```

```
-100
<class 'int'>
-101
<class 'int'>
```

Why -100, not -101?
Round vs Truncation

Type Casting

```
int('i am sleepy')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-22-087586fc6edb> in <cell line: 1>()
----> 1 int('i am sleepy')

ValueError: invalid literal for int() with base 10: 'i am sleepy'
```

Next steps: Explain error

Arithmetic Operators

Operator	Name	Example
+	Addition	
-	Subtraction	
*	Multiplication	
/	Division	10/0 ZeroDivisionError **4/2 integer or float?**
//	Floor Division	13.9//2 = 6.0
%	Modulus	11%3 = 2; 11.0%3.0 = 2.0
**	Exponent	2**4 = 16
+=	Augmented Addition Operator	a+=1 equals to a = a+1
-=		
*=		
…		

Review

- Variables
  - Id
  - Name
  - Type
  - Type Casting
  - Arithmetic operators

Outline

- Module introduction
- Python Introduction
- Some concept of software engineering
- Python basic 1
- **Some concept of programming**

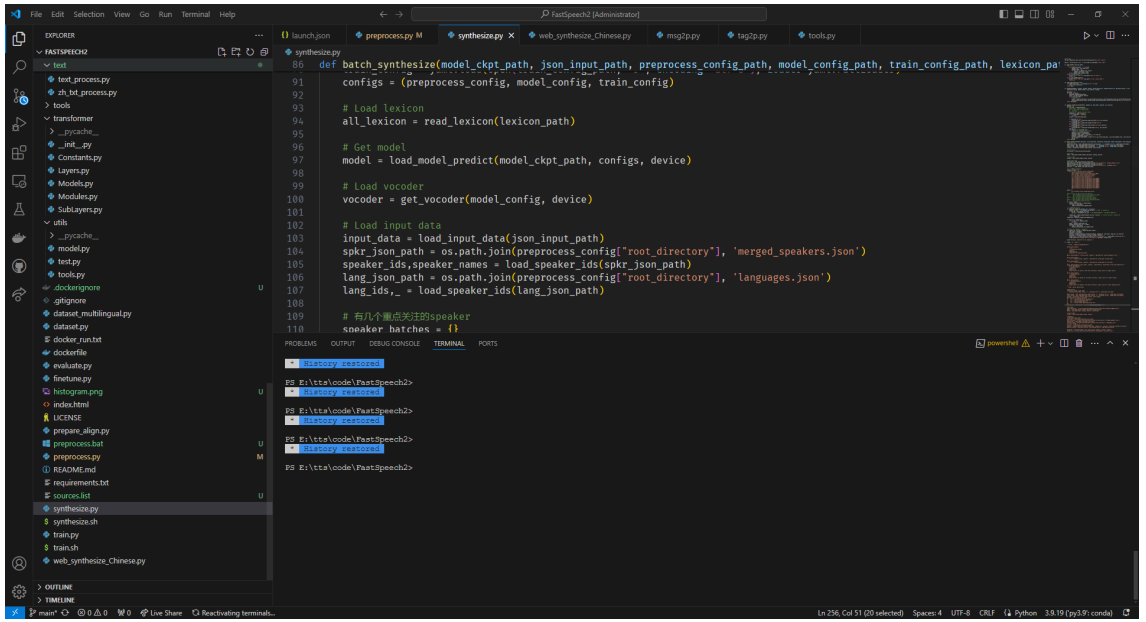
Command Line and IDE

- Command line - Python Shell
  - Interactive programming Python shell
  - Show the results immediately after the statements
  - Cons:
    - Difficult to manage in large projects
    - Difficult to refactor

```
(py3.9) C:\Users\A103624>python
Python 3.9.19 (main, May  6 2024, 20:12:36) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print('1234')
1234
>>>
```

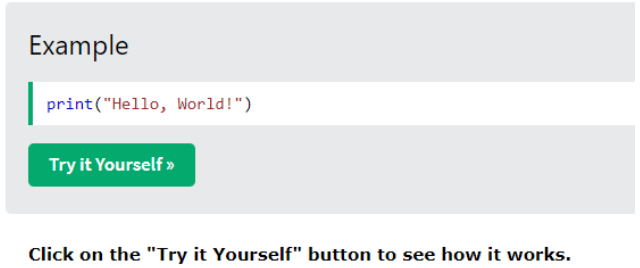
Command Line and IDE

- Integrated Development Environments
  - A programming environment
    - A code editor
    - A compiler
    - A debugger
    - A graphical user interface (GUI) builder



Command Line and IDE

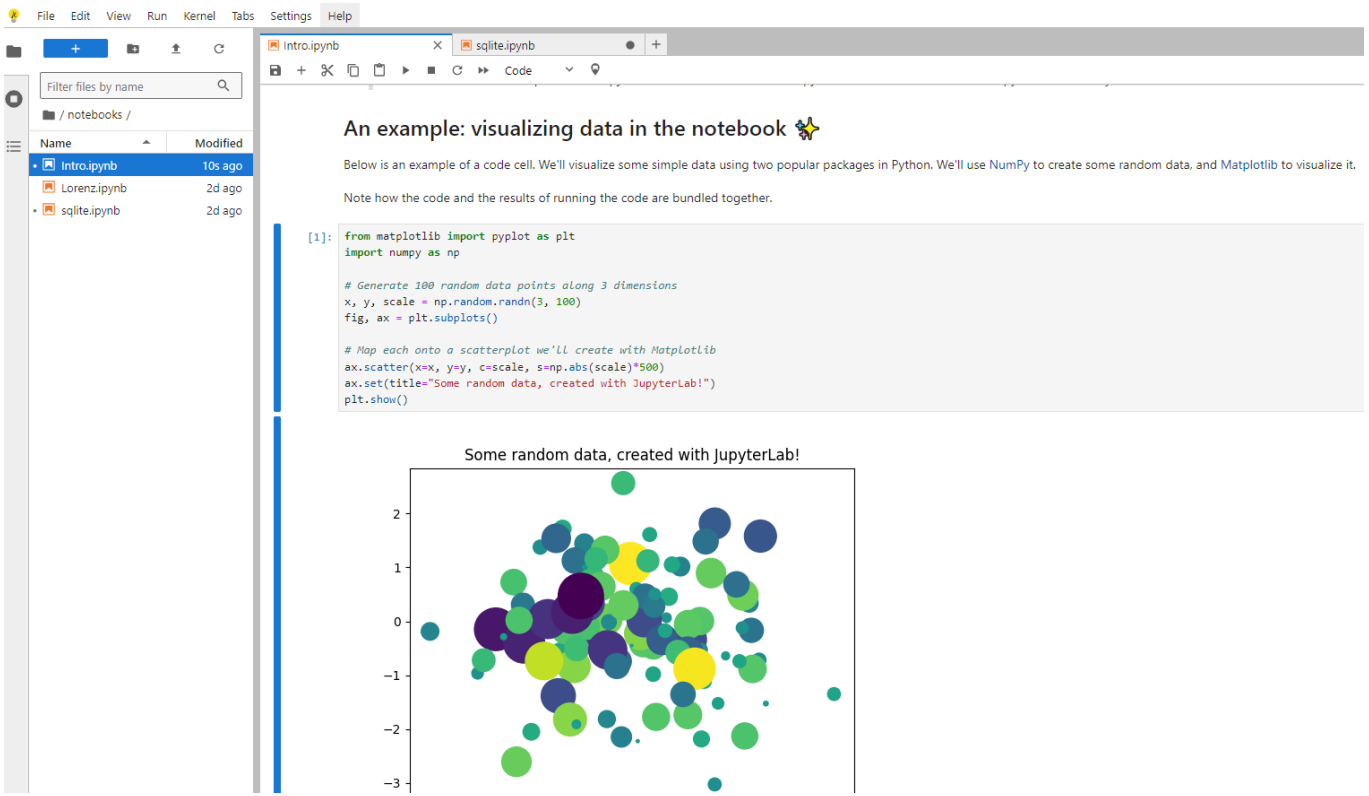
- https://www.w3schools.com/python/



- https://colab.research.google.com/
  - We will use this one in class
  - Save your results easily
  - **Use the auto-complete function properly**

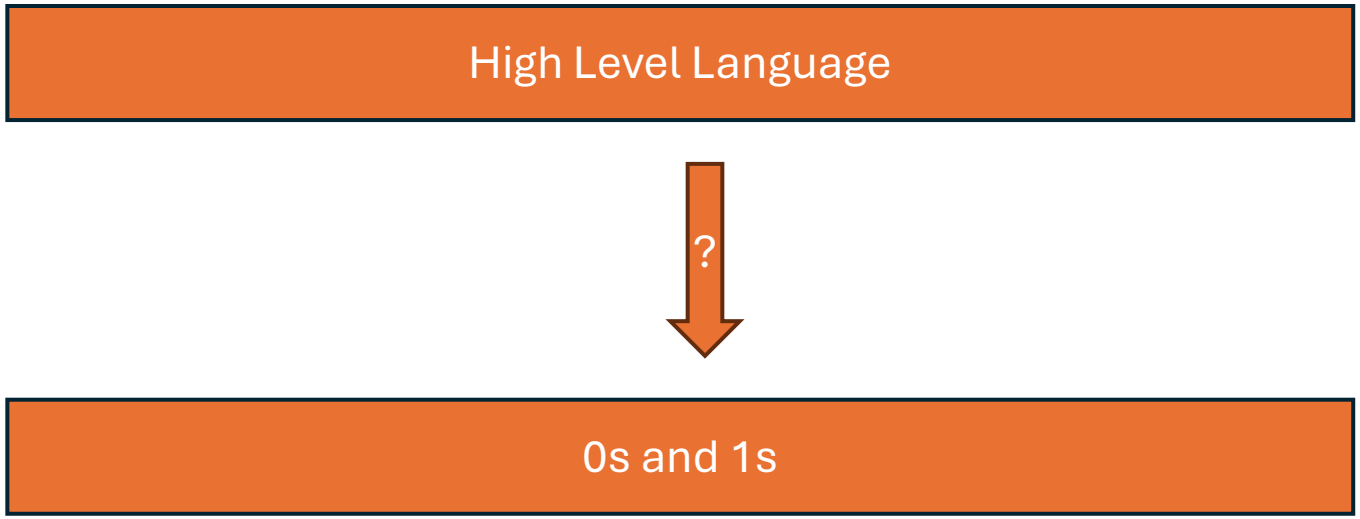
Jupyter Notebook

- https://jupyter.org/
- https://jupyter.org/try-jupyter/lab/



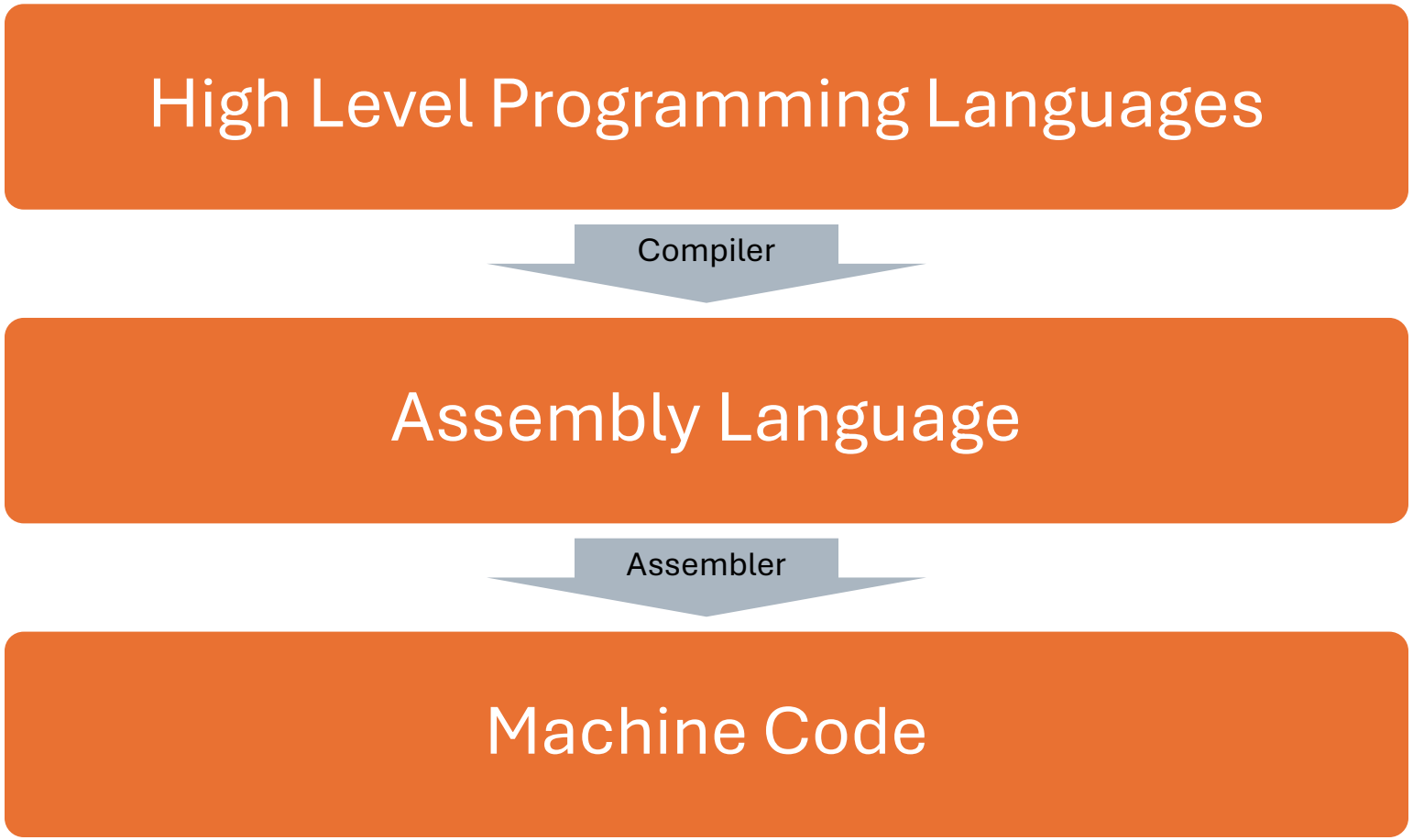
What is programming?

- Computer programming or coding is the composition of sequences of instructions, called programs, that computers can follow to perform tasks.

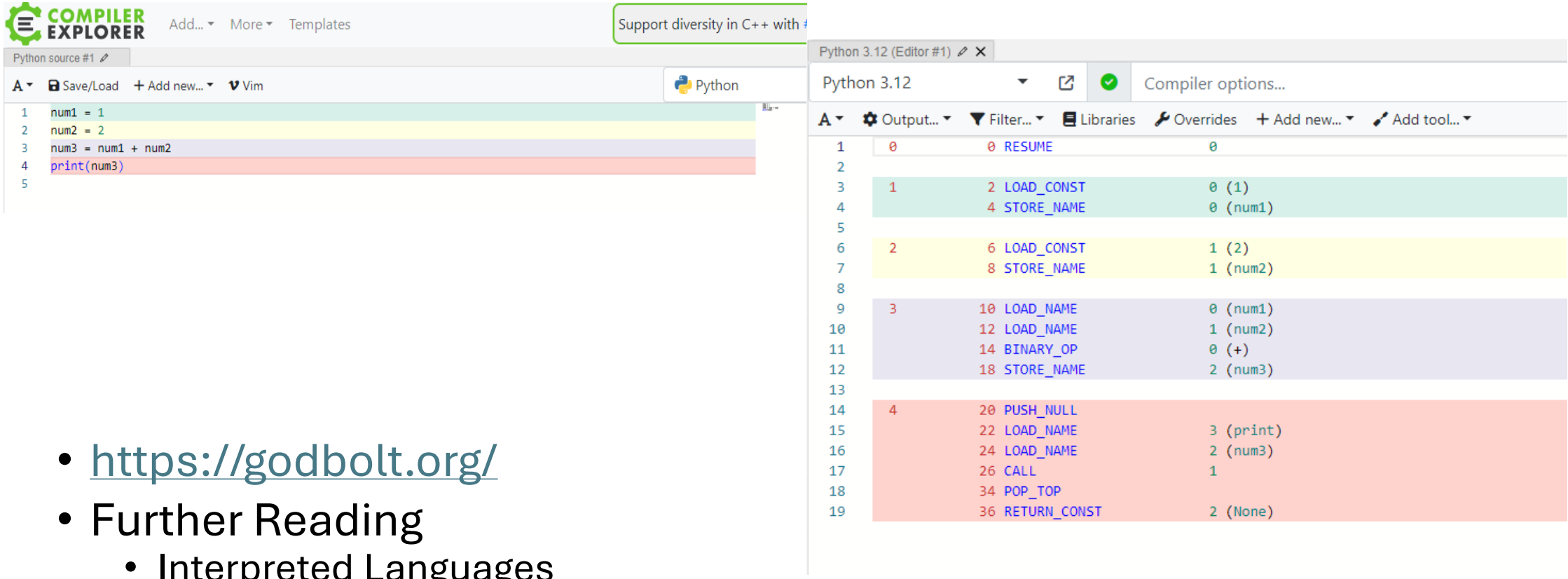


https://en.wikipedia.org/wiki/Computer_programming

From high level code to 0 and 1 (Compiled Language)



Bytecode



- https://godbolt.org/
- Further Reading
  - Interpreted Languages
  - Compiled Languages

Bytecode

0s

```
import dis

def example():
    num1 = 1
    num2 = 2
    return num1 + num2

dis.dis(example)
```

```
  4           0 LOAD_CONST               1 (1)
              2 STORE_FAST               0 (num1)

  5           4 LOAD_CONST               2 (2)
              6 STORE_FAST               1 (num2)

  6           8 LOAD_FAST                0 (num1)
             10 LOAD_FAST                1 (num2)
             12 BINARY_ADD
             14 RETURN_VALUE
```

CPython

- CPython is an interpreter responsible for reading, parsing, and executing Python code.
  - Translate high level code to bytecode
  - Execute the bytecode

Compiler and Interpreter

- Compiler
  - Translates the entire source code of a program into machine code before execution.
  - Generates an executable file that can be run independently of the source code.
  - Usually faster execution after compilation because the translation is done once.
  - Error detection happens at compile-time, making debugging easier before running the program.
  - Examples: GCC (GNU Compiler Collection), MSVC (Microsoft Visual C++), Clang.

- Interpreter
  - Translates and executes the source code line by line at runtime.
  - Does not produce an independent executable file; the source code must be present during execution.
  - Slower execution because translation happens every time the program is run.
  - Immediate execution allows for easier testing and debugging of small code snippets.
  - Examples: Python interpreter, Ruby interpreter, JavaScript engines (like V8).

Review

- Command Line and IDE
- High Level Code
- Bytecode / Assembly Language
- Machine Code

Assignment

- Assignment
  - Install the Python 3
    - Try pip install
    - Conda / Miniconda
    - Install Conda
    - Create an environment
  - Install the IDE that you like to use (i.e., Pycharm , Visual Studio Code)
    - Create a simple program
    - Try run, debug, breakpoint, check the variant value

- First lab from week 2

Good Programming Practices

Bjarne Stroustrup, inventor of C++ and author of *The C++ Programming Language*

I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well.

Martin R C. Clean code: a handbook of agile software craftsmanship[M]. Pearson Education, 2009.