CUHK Business School
The Chinese University
of Hong Kong

MSC IN FINANCE
PRE-TERM COURSE WEEK1:
Basic Programming With Python

Tutor: Yuan Lu
CUHK Department of Finance
yuan.lu@link.cuhk.edu.hk

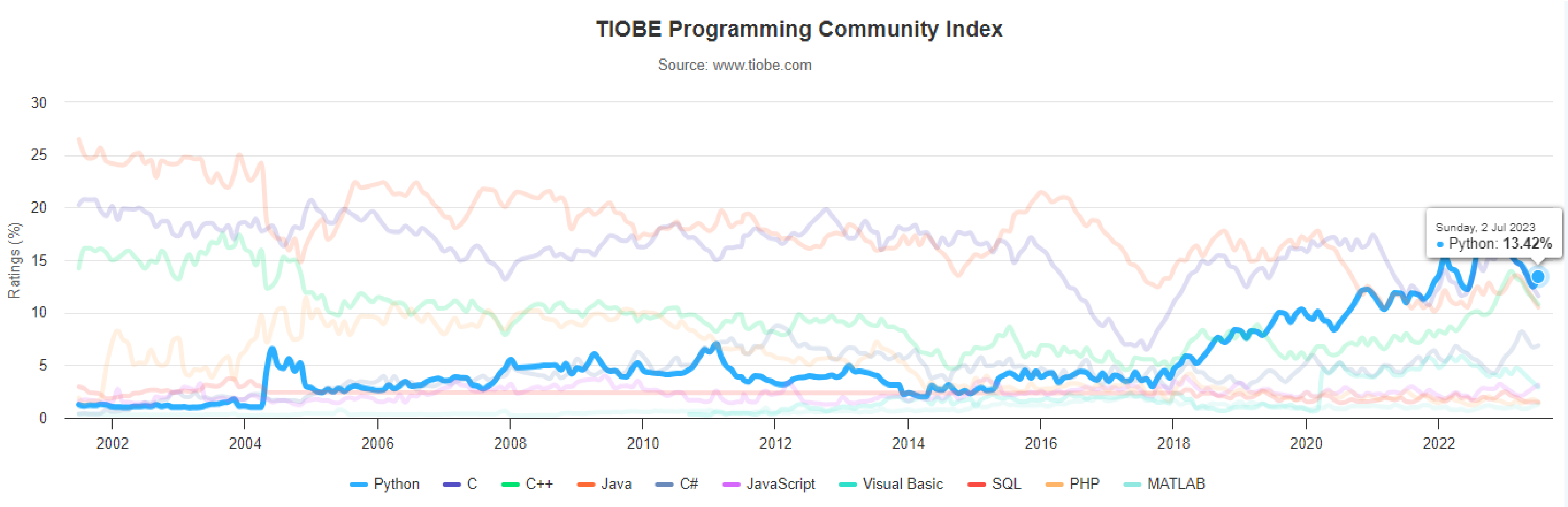
July 18, 2023

Agenda:

1. Why Python?
2. Getting Started!
3. Basic Data Types and Data Structures.
4. Programming Structures.
5. Functions.

Why Python?

- The Most Popular programming languages based on TIOBE Programming Community index.



Why Python? (Cont.)

- Python enables programs to be written compactly and readably.
  - The high-level data types;
  - Indentation instead of brackets;
  - No variable or argument declarations are necessary.

Why Python? (Cont.)

- Python enables programs to be written compactly and readably.

```python
def ex1(A):
    sum = A[0]
    for i in A:
        sum = sum + A[i]
    return sum
```

Pros of Python v.s. C++

- Shorter.
- No declaration.
- Automatically memory management.

```cpp
double ex1(double A[], size_t n)
{
    double sum = A[0];
    for(size_t i = 0; i < n; i++)
    {
        s = s + A[i];
    }
    return sum;
 }
```

Why Python? (Cont.)

- Extensive standard libraries are freely available.
  - Pandas, Numpy, Keras, TensorFlow, Scikit Learn, and etc.

- Linear Regressions in Python v.s. in C++ (on the next page)

```
from sklearn.linear_model import LinearRegression

# Data points
X = [[1], [2], [3], [4], [5]]
y = [2, 4, 5, 4, 5]

# Create a linear regression object
model = LinearRegression()

# Fit the linear regression model to the data
model.fit(X, y)

# Print the slope and intercept
print('Slope:', model.coef_[0])
print('Intercept:', model.intercept_)
```

Why Python? (Cont.)

Pros of Python v.s. C++

- Simple and beginner-friendly.
- Quickly to implement your ideas based on various open resources and packages.

```cpp
#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

// Function to calculate the mean of a vector
double mean(vector<double> v) {
double sum = 0.0;
for (int i = 0; i < v.size(); i++) {
sum += v[i];
}
return sum / v.size();
}
```

```cpp
// Function to calculate the slope and intercept of a
linear regression line
void linearRegression(vector<double> x, vector<double> y,
double &slope, double &intercept) {
double x_mean = mean(x);
double y_mean = mean(y);
double numerator = 0.0;
double denominator = 0.0;
for (int i = 0; i < x.size(); i++) {
numerator += (x[i] - x_mean) * (y[i] - y_mean);
denominator += pow(x[i] - x_mean, 2);
}
slope = numerator / denominator;
intercept = y_mean - slope * x_mean;
}

int main() {
vector<double> x = {1.0, 2.0, 3.0, 4.0, 5.0};
vector<double> y = {2.0, 4.0, 5.0, 4.0, 5.0};
double slope, intercept;
linearRegression(x, y, slope, intercept);
cout << "Slope: " << slope << endl;
cout << "Intercept: " << intercept << endl;
return 0;
}
```

Why Python? (Cont.)

- An easy to learn, effective and powerful programming language.
- Object-oriented, with elegant syntax and dynamic typing.
- Extensive standard library are freely available.
- Well-connected and supportive community.
- Compared to other popular programming languages (C/C++, Java), Python is beginner-friendly, allowing us to quickly convert ideas into working code.
- Compared to conventional analytical tools (Excel VBA) in finance, Python is faster, more flexible, and more efficient, enabling real-time analysis, big data analysis, and AI-related productivity.

Getting Started with Python

IDE: Integrated Development Environment.

- **Jupyter notebook**
    - Easily present and share data visualizations along with code and text.
- **Spyder**
    - Build lightweight data science applications with multiple scripts or multiprocessing.

Install via **Anaconda**
https://www.anaconda.com/

Other IDE:

- **PyCharm** https://www.jetbrains.com/pycharm/
    - Work on build large, complex projects.
- **VS code** https://code.visualstudio.com/
    - Work with multiple languages and customized IDE.

Getting Started with Python (Cont.)

For Beginners
If you have absolutely no experience with Python or coding in general, below is a short introduction to why and how we're going to use Python (timestamped to start at 1 minute 18 seconds, watch until 4:19).

Getting Python (Using Python on Your Local Machine)
Anaconda installation is the recommended method for getting Python. Anaconda is a package manager that allows installing many applications at once. Among other applications, Anaconda also installs Jupyter notebook, an application where you can easily write and execute Python codes. Pick either Video Guide or Text Guide.

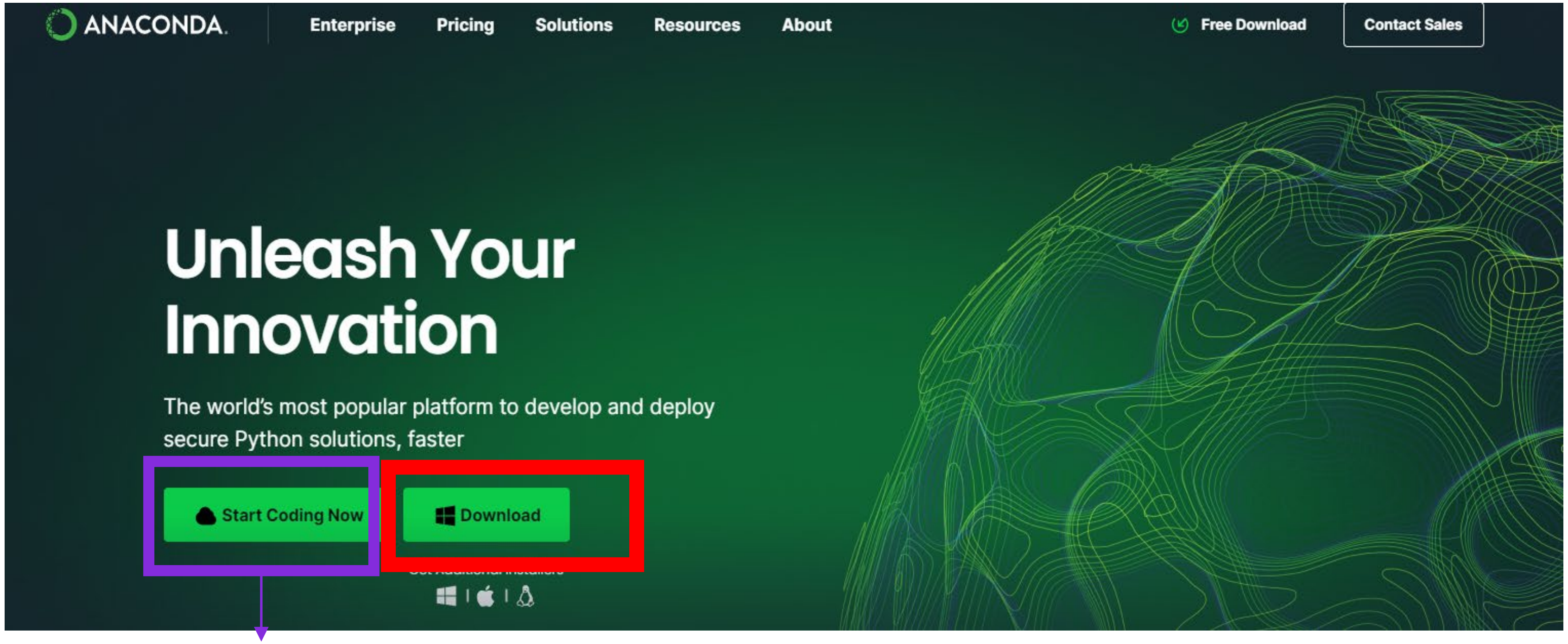
- **Video Guide**
- https://youtu.be/Z1Yd7upQsXY?t=4m19s (timestamped to start minutes 19 seconds watch until 5:59)
- **Text Guide**
- https://docs.anaconda.com/anaconda/install/
  —How to install Anaconda for various platforms, see for an explanation of Anaconda.
- https://docs.anaconda.com/anaconda/navigator/
  —How to use Anaconda Navigator, a GUI interface for Anaconda.
- https://docs.anaconda.com/anaconda/user-guide/getting-started/#run-python-in-a-jupyter-notebook
  —Launching Jupyter Notebook through Anaconda Navigator, see For Absolute Beginners for an explanation of Jupyter Notebook.

Getting Started with Python (Cont.)

Download Anaconda from their website: https://www.anaconda.com/download

Proceed the installing step by step:

- Windows: https://docs.anaconda.com/free/anaconda/install/windows/
- MacOS: https://docs.anaconda.com/free/anaconda/install/windows/



Unleash Your Innovation

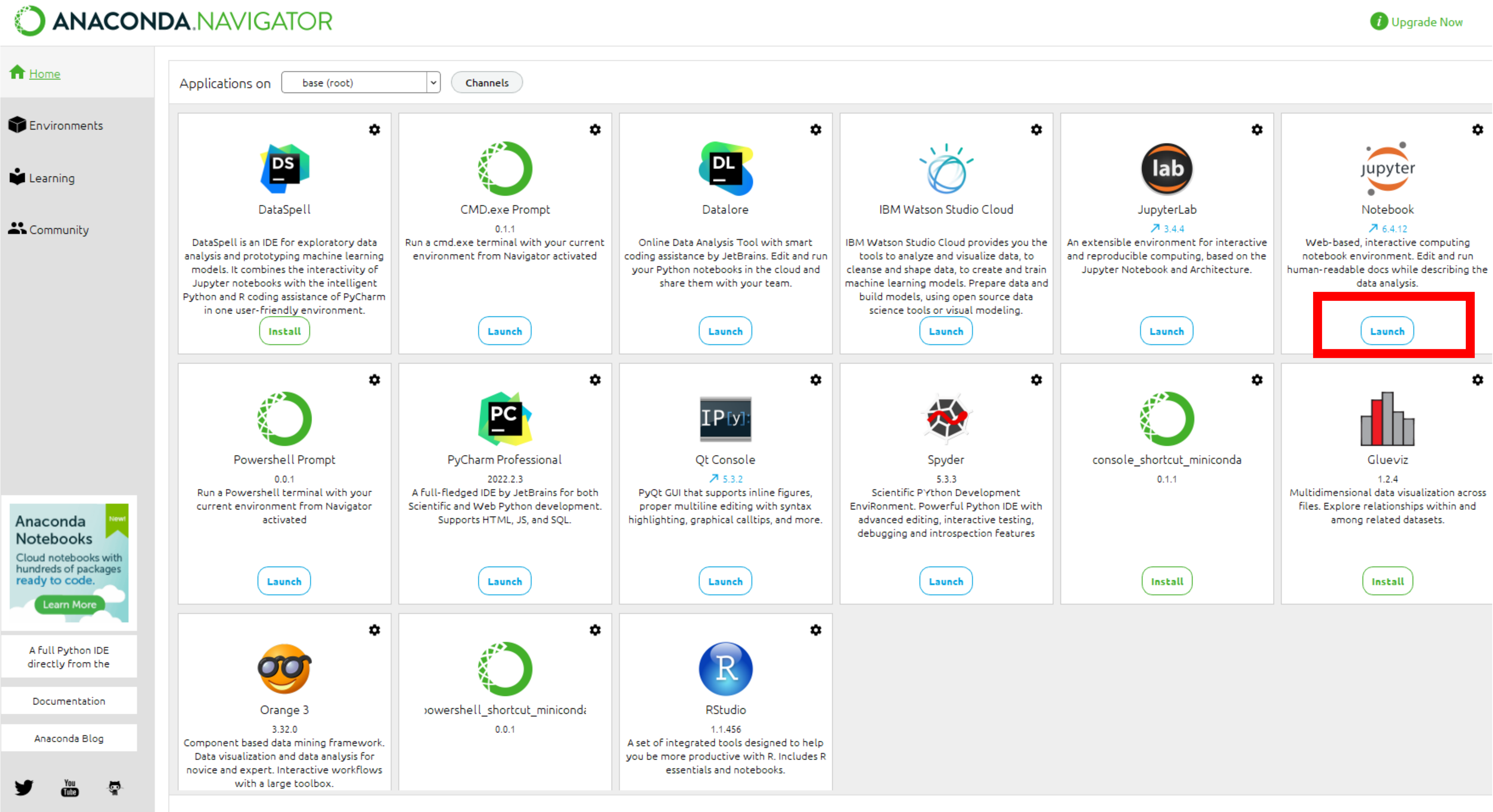
The world's most popular platform to develop and deploy secure Python solutions, faster

Run Jupyter online.

Or use Google Colab to run your codes on clouds.

But in the course, we still recommend you install Jupyter Notebook in your machine.

Getting Started with Python (Cont.)



DataSpell

DataSpell is an IDE for exploratory data analysis and prototyping machine learning models. It combines the interactivity of Jupyter notebooks with the intelligent Python and R coding assistance of PyCharm in one user-friendly environment.

CMD.exe Prompt

0.1.1

Run a cmd.exe terminal with your current environment from Navigator activated

Datalore

Online Data Analysis Tool with smart coding assistance by JetBrains. Edit and run your Python notebooks in the cloud and share them with your team.

IBM Watson Studio Cloud

IBM Watson Studio Cloud provides you the tools to analyze and visualize data, to cleanse and shape data, to create and train machine learning models. Prepare data and build models, using open source data science tools or visual modeling.

JupyterLab

↗ 3.4.4

An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.

Notebook

↗ 6.4.12

Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

Powershell Prompt

0.0.1

Run a Powershell terminal with your current environment from Navigator activated

PyCharm Professional

2022.2.3

A full-fledged IDE by JetBrains for both Scientific and Web Python development. Supports HTML, JS, and SQL.

Qt Console

↗ 5.3.2

PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.

Spyder

5.3.3

Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features

console_shortcut_miniconda

0.1.1

Glueviz

1.2.4

Multidimensional data visualization across files. Explore relationships within and among related datasets.

Orange 3

3.32.0

Component based data mining framework. Data visualization and data analysis for novice and expert. Interactive workflows with a large toolbox.

owershell_shortcut_minicond

0.0.1

RStudio

1.1.456

A set of integrated tools designed to help you be more productive with R. Includes R essentials and notebooks.

Anaconda Notebooks

Cloud notebooks with hundreds of packages ready to code.

A Full Python IDE directly from the

Documentation

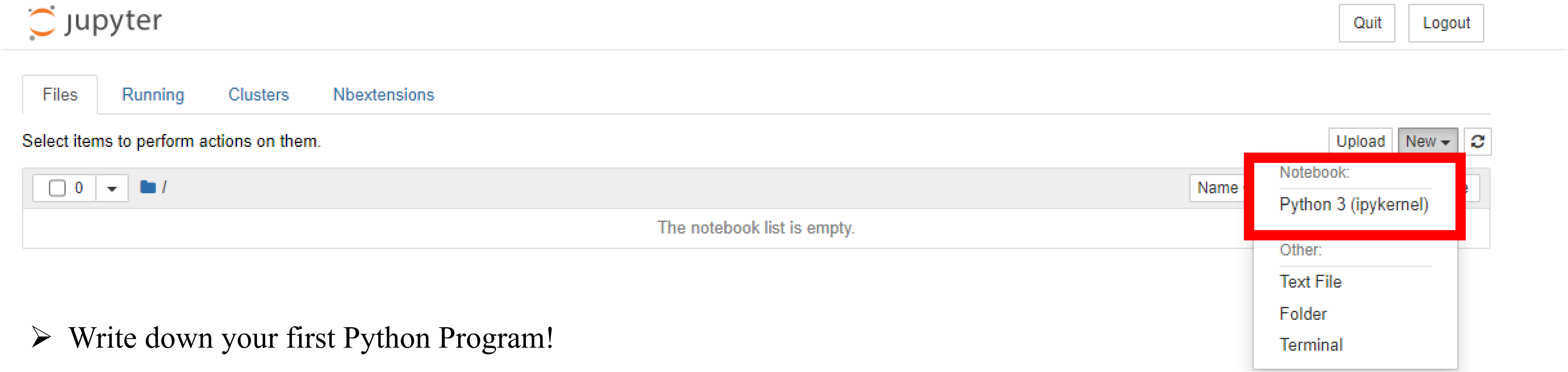
Anaconda Blog

Anaconda Navigator:

- Launch Jupyter Notebook

Getting Started with Python (Cont.)

- Generate a new Python 3 Notebook on Jupyter Notebook

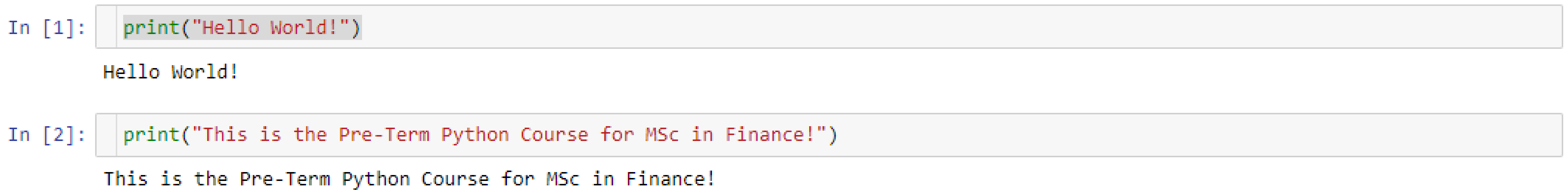


Select items to perform actions on them.

□ 0 ▾ /

The notebook list is empty.

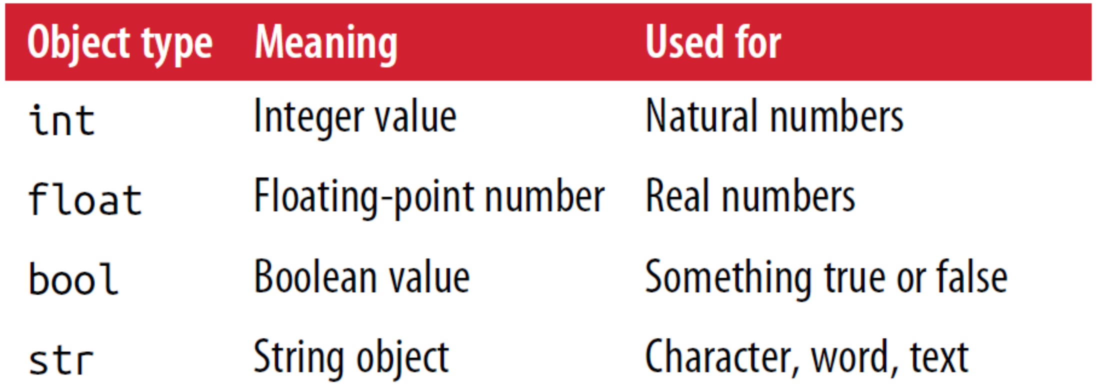
- Write down your first Python Program!



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

Hello World!

In [2]: print("This is the Pre-Term Python Course for MSc in Finance!")

This is the Pre-Term Python Course for MSc in Finance!
```

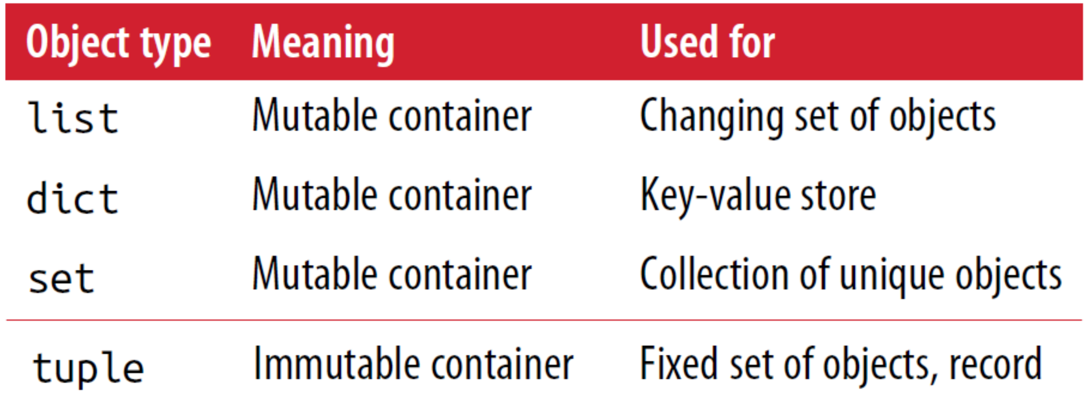
Basic Data Types and Data Structures

Data Types:



Object type	Meaning	Used for
`int`	Integer value	Natural numbers
`float`	Floating-point number	Real numbers
`bool`	Boolean value	Something true or false
`str`	String object	Character, word, text

Data Structures:

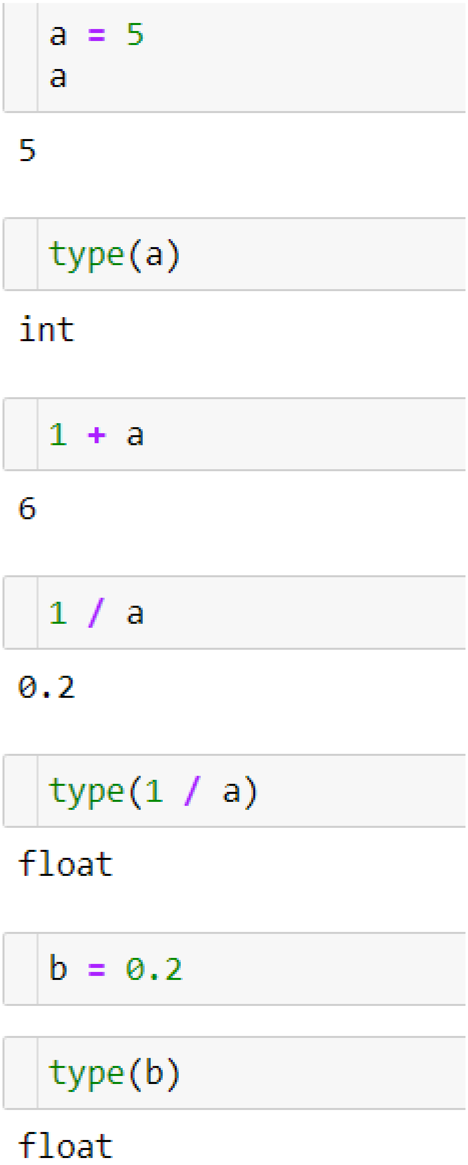


Object type	Meaning	Used for
`list`	Mutable container	Changing set of objects
`dict`	Mutable container	Key-value store
`set`	Mutable container	Collection of unique objects
`tuple`	Immutable container	Fixed set of objects, record

Reference: Yves Hilpisch (2018), Python for Finance: Mastering Data-Driven Finance (2nd Edition), O'Reilly.

Basic Data Types: *Integer, Float and Bool*

Try to assign values to variables and check their data types.



```
a = 5
a
```

```
5
```

```
type(a)
```

```
int
```

```
1 + a
```

```
6
```

```
1 / a
```

```
0.2
```

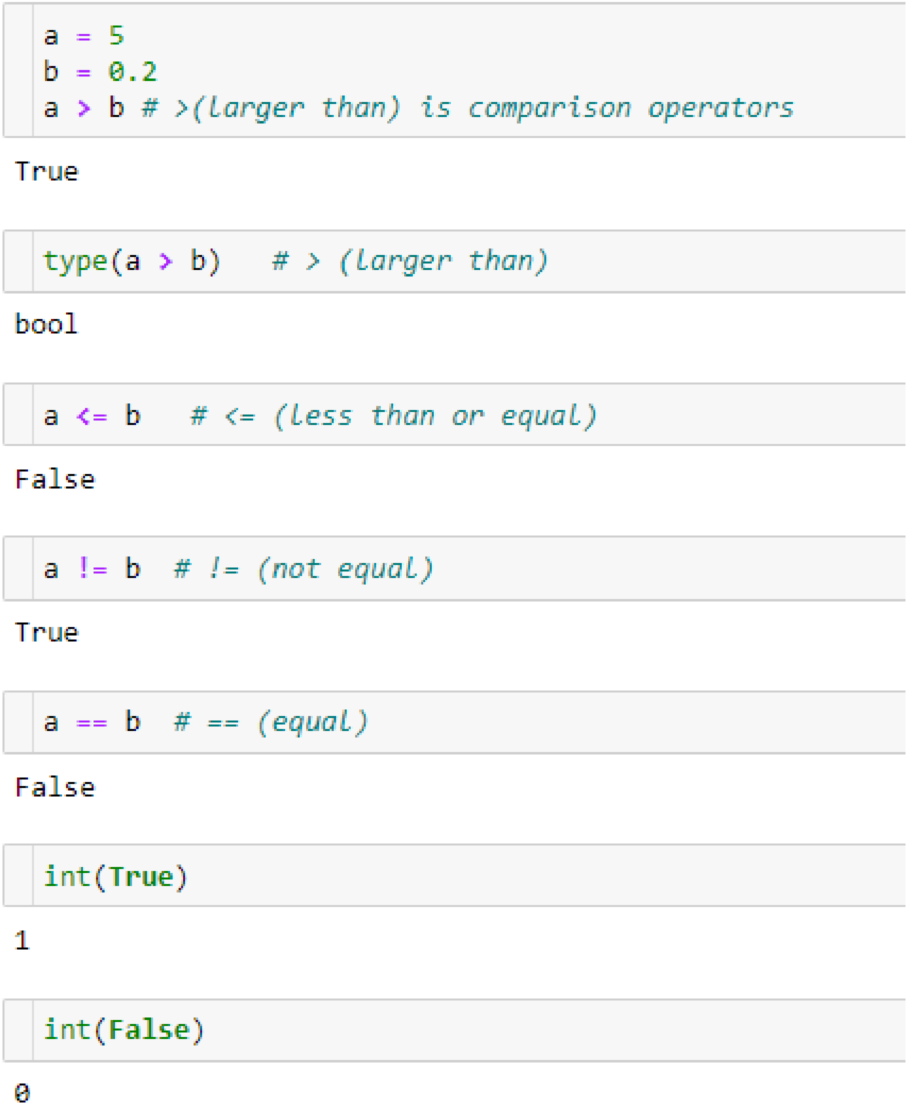
```
type(1 / a)
```

```
float
```

```
b = 0.2
```

```
type(b)
```

```
float
```



```
a = 5
b = 0.2
a > b # >(larger than) is comparison operators
```

```
True
```

```
type(a > b)    # > (larger than)
```

```
bool
```

```
a <= b    # <= (less than or equal)
```

```
False
```

```
a != b   # != (not equal)
```

```
True
```

```
a == b   # == (equal)
```

```
False
```

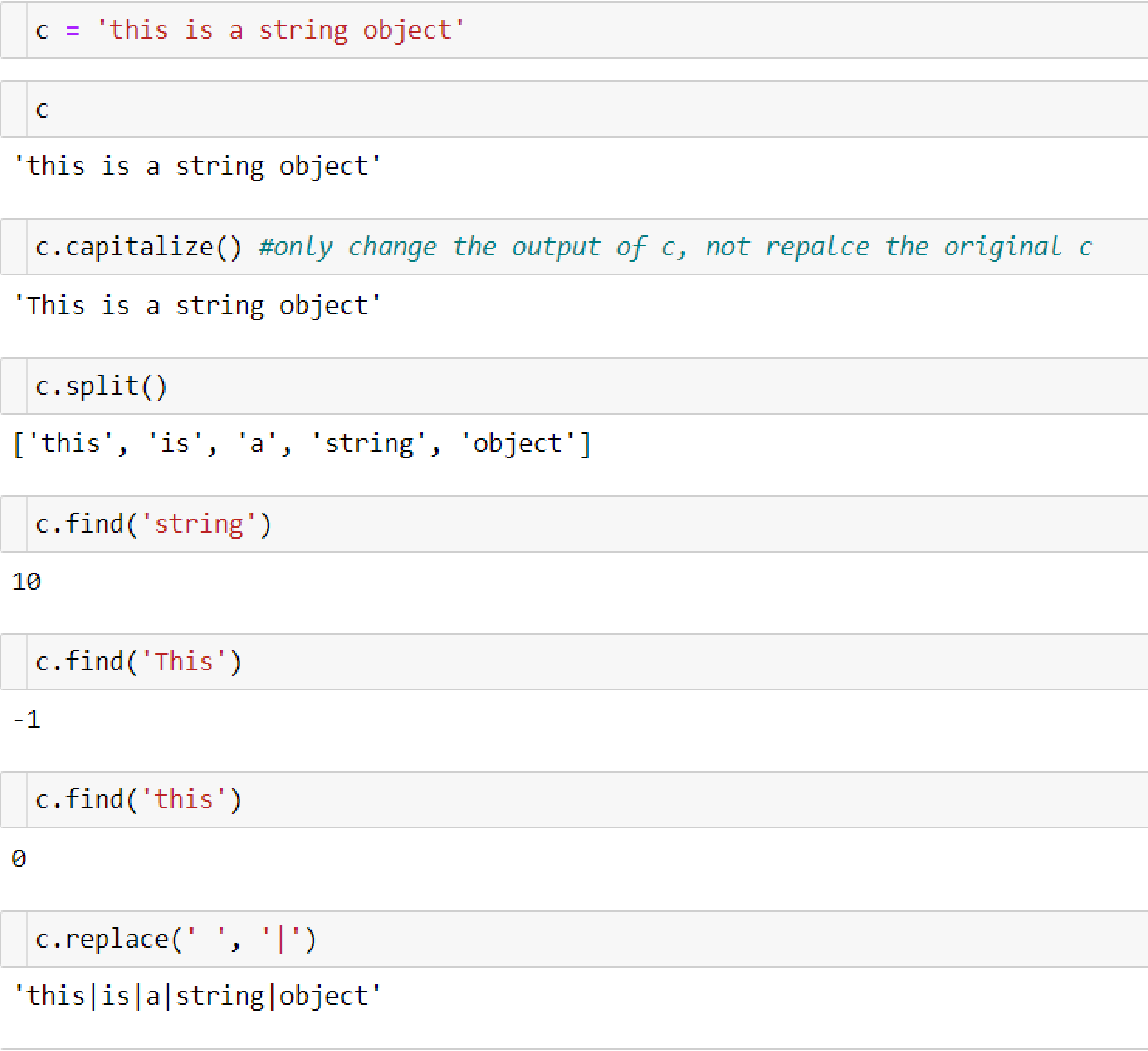
```
int(True)
```

```
1
```

```
int(False)
```

```
0
```

Basic Data Types: *String*



```
c = 'this is a string object'
```

```
c
```

```
'this is a string object'
```

```
c.capitalize() #only change the output of c, not repalce the original c
```

```
'This is a string object'
```

```
c.split()
```

```
['this', 'is', 'a', 'string', 'object']
```

```
c.find('string')
```

```
10
```

```
c.find('This')
```

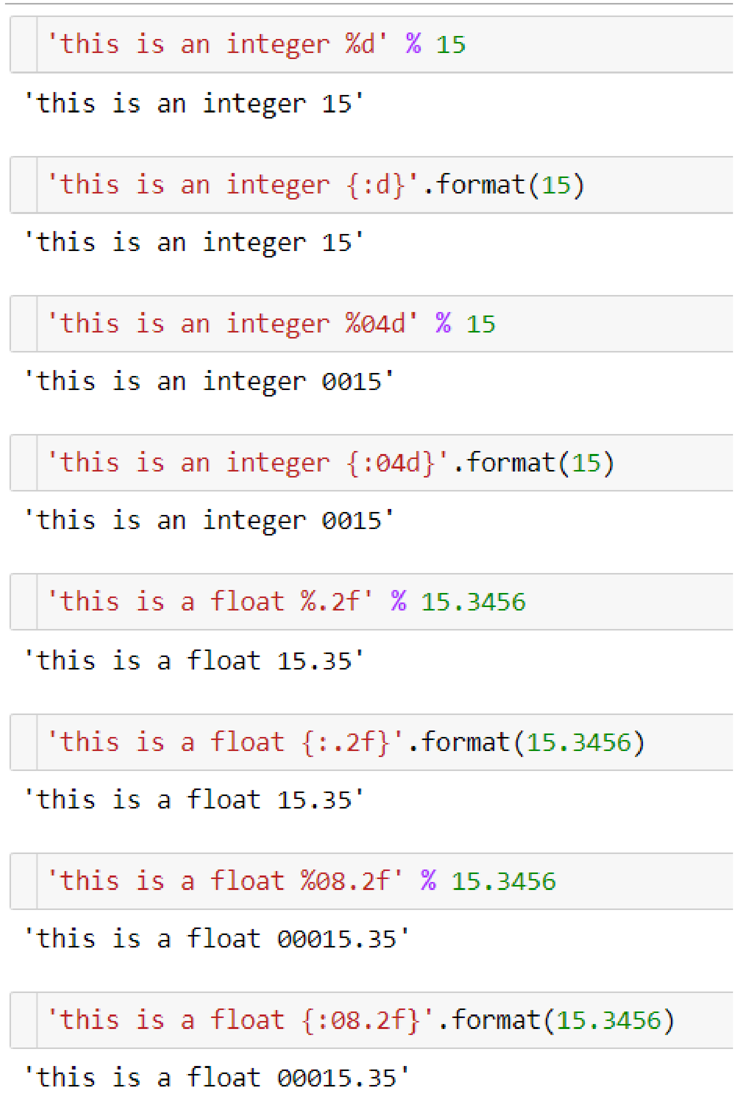
```
-1
```

```
c.find('this')
```

```
0
```

```
c.replace(' ', '|')
```

```
'this|is|a|string|object'
```



```
'this is an integer %d' % 15
```

```
'this is an integer 15'
```

```
'this is an integer {:d}'.format(15)
```

```
'this is an integer 15'
```

```
'this is an integer %04d' % 15
```

```
'this is an integer 0015'
```

```
'this is an integer {:04d}'.format(15)
```

```
'this is an integer 0015'
```

```
'this is a float %.2f' % 15.3456
```

```
'this is a float 15.35'
```

```
'this is a float {:.2f}'.format(15.3456)
```

```
'this is a float 15.35'
```

```
'this is a float %08.2f' % 15.3456
```

```
'this is a float 00015.35'
```

```
'this is a float {:08.2f}'.format(15.3456)
```

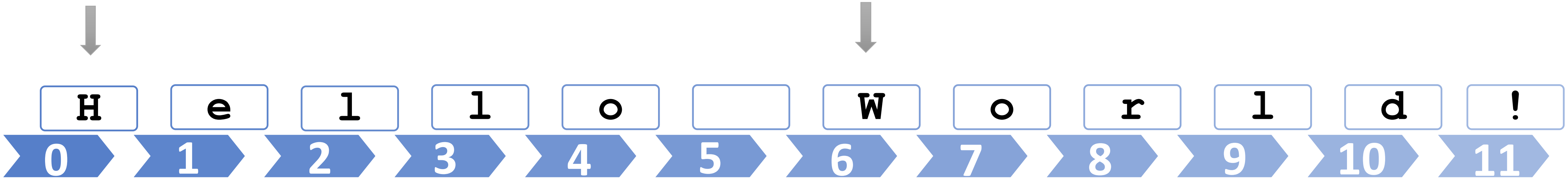
```
'this is a float 00015.35'
```

Basic Data Types: *String (Cont.)– Extracting Elements*

➢ Indexing

```
print(b[0])
→ 'H'
```

- ✓ Python indexing starts at 0
- ✓ Index operator [ ] is used for indexing
- ✓ Think of index as a pointer between the elements



H	e	l	l	o		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11

```
print(c.index('W'))
→ 6
```

Basic Data Types: *String (Cont.)– Extracting Elements*

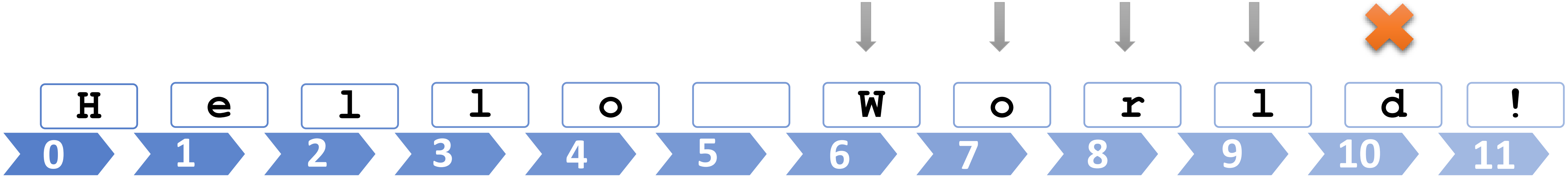
➢ Slicing

```
  print(b[6:10])
→ 'Worl'
```

What is <u>after the start</u> index?

What is <u>before the stop</u> index?

- ✓ Index operator with a colon [:] is used for slicing
- ✓ Start and stop index can be defined on either side of the colon
- ✓ Stop index defines where to stop extracting elements



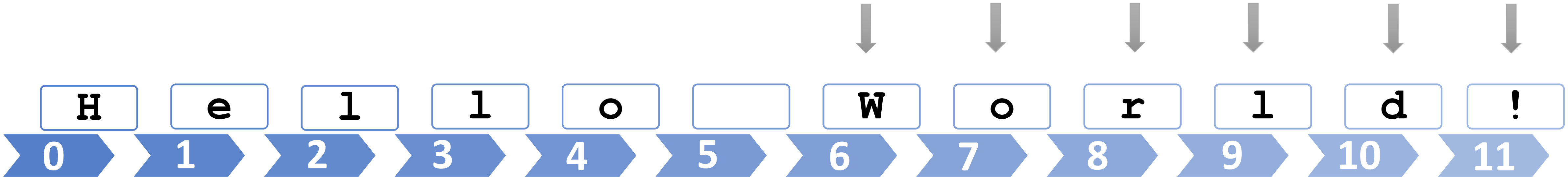
H	e	l	l	o		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11

Basic Data Types: *String (Cont.)– Extracting Elements*

➢ Slicing

```
print(b[6: ])
→'World!'
```

Everything after the start index



H	e	l	l	o		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11

Basic Data Types: *String (Cont.)– Extracting Elements*

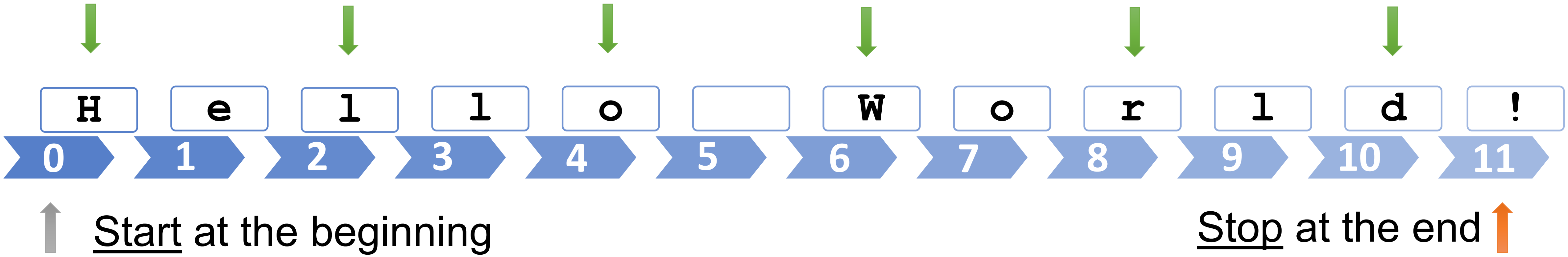
Stepping

```
print(b[::2 ])
```

→ 'HloWrd'

✓ Index operator with double colon [: :] is used for stepping

Select every other element between start and stop indices



H	e	l	l	o		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11

Start at the beginning

Stop at the end

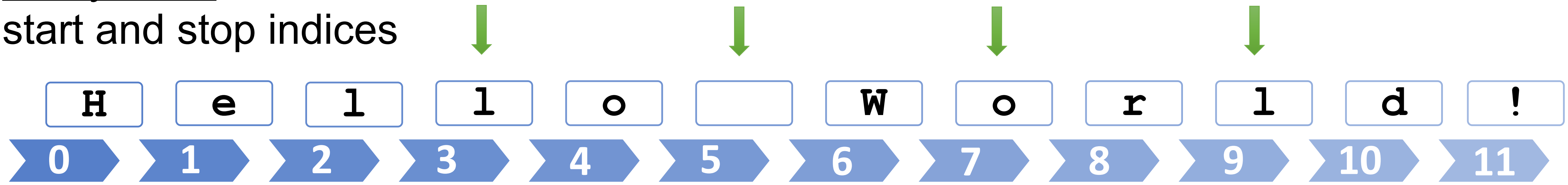
Basic Data Types: *String (Cont.)– Extracting Elements*

➢ Slicing with Stepping

```
  print(b[3:10:2])
→'l ol'
```

✓ Start, stop and step index can be defined on each sides of the colons

<u>Every other</u> element between start and stop indices



H	e	l	l	o		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11

What is <u>after the start</u> index?

What is <u>before the stop</u> index?

Basic Data Types: *String (Cont.)– Extracting Elements*

➢ Slicing with Stepping

```
print(b[10:3:2])
→ ''
```

✓ We can use negative index to go backwards

Does NOT work because elements are extracted from right of the start position up to the left of stop position

H	e	l	l	o		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11

↑ Stop before start ?? ✗

After the start ↑

Basic Data Types: *String (Cont.)– Extracting Elements*

➢ Negative Indexing

```
  print(b[-12])
→'H'
```

✓ Negative index does not have 0

H	e	l	l	o		W	o	r	l	d	!
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

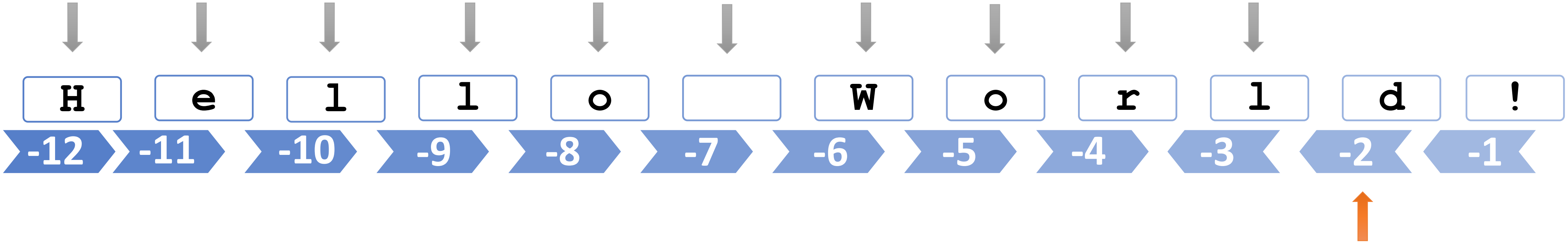
```
  print(b[-1])
→ '!'
```

Basic Data Types: *String (Cont.)– Extracting Elements*

➢ Slicing with Negative Index

```
  print(b[:-2])
→'Hello Worl'
```

Everything except last two items

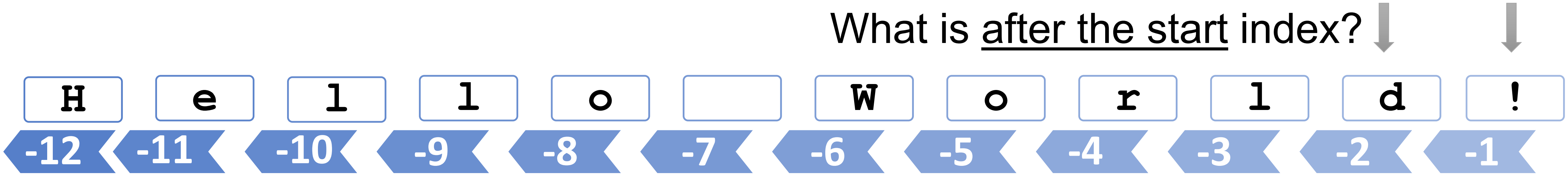


Basic Data Types: *String (Cont.)– Extracting Elements*

➢ Slicing with Negative Index

```
  print(b[-2:])
→'d!'
```

Everything after start



What is after the start index?

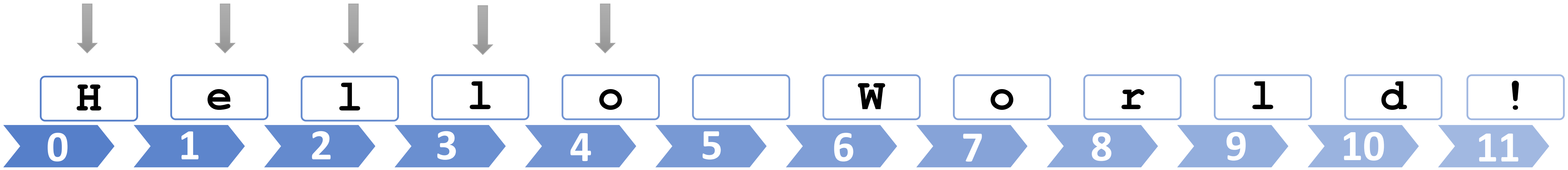
H	e	l	l	o		W	o	r	l	d	!
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Basic Data Types: *String (Cont.)– Extracting Elements*

➢ Slicing and Stepping with Negative Index

```
  print(b[4::-1])
→'olleH'
```

First five elements, <u>reversed</u>



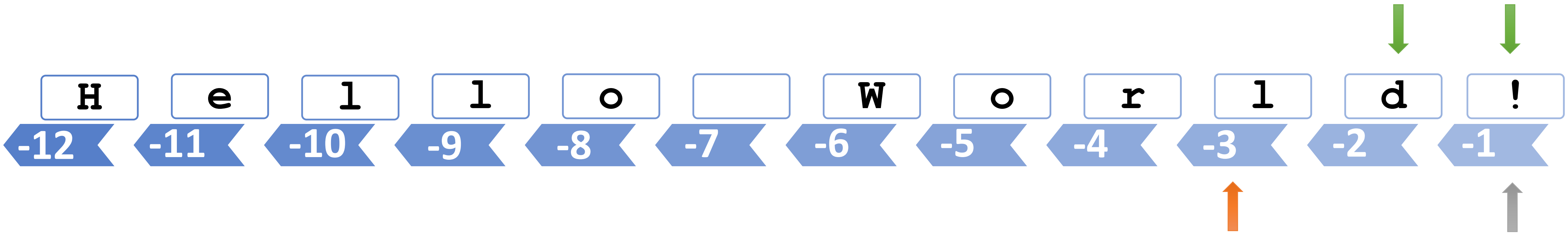
H	e	l	l	o		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11

Basic Data Types: *String (Cont.)– Extracting Elements*

➢ Slicing and Stepping with Negative Index

```
  print(b[:-3:-1])
→'!d'
```

Last two items, reversed



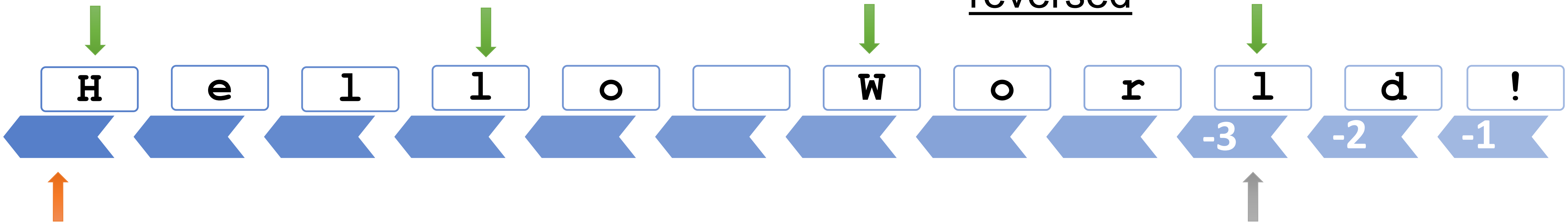
H	e	l	l	o		W	o	r	l	d	!
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Basic Data Types: *String (Cont.)– Extracting Elements*

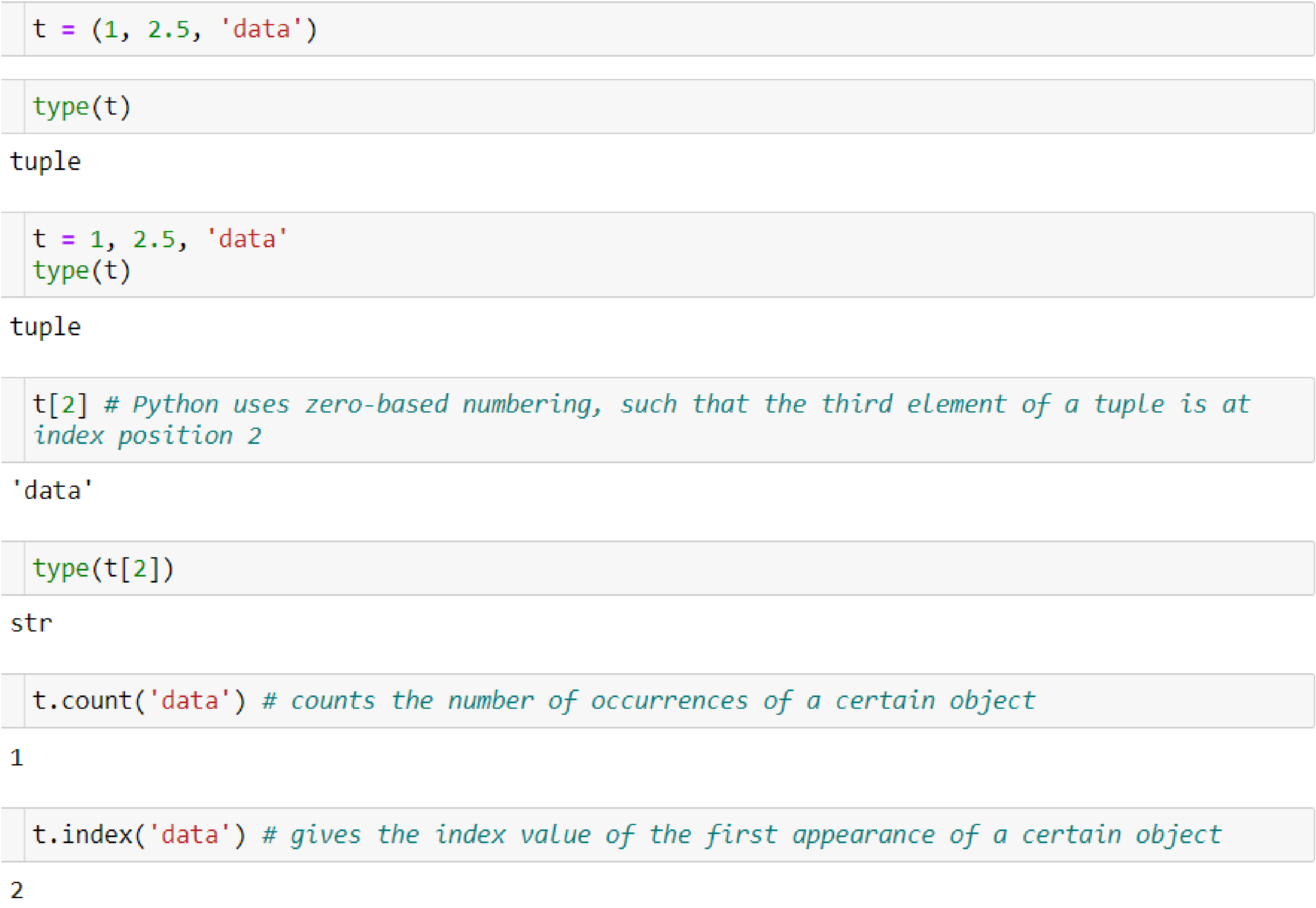
➢ Slicing and Stepping with Negative Index

```
  print(b[-3::-3])
→'lWlH'
```

Every two other element,
except the last two elements,
reversed



Basic Data Structures: *Tuple*



```
t = (1, 2.5, 'data')
```

```
type(t)
```

```
tuple
```

```
t = 1, 2.5, 'data'
type(t)
```

```
tuple
```

```
t[2] # Python uses zero-based numbering, such that the third element of a tuple is at
index position 2
```

```
'data'
```

```
type(t[2])
```

```
str
```

```
t.count('data') # counts the number of occurrences of a certain object
```

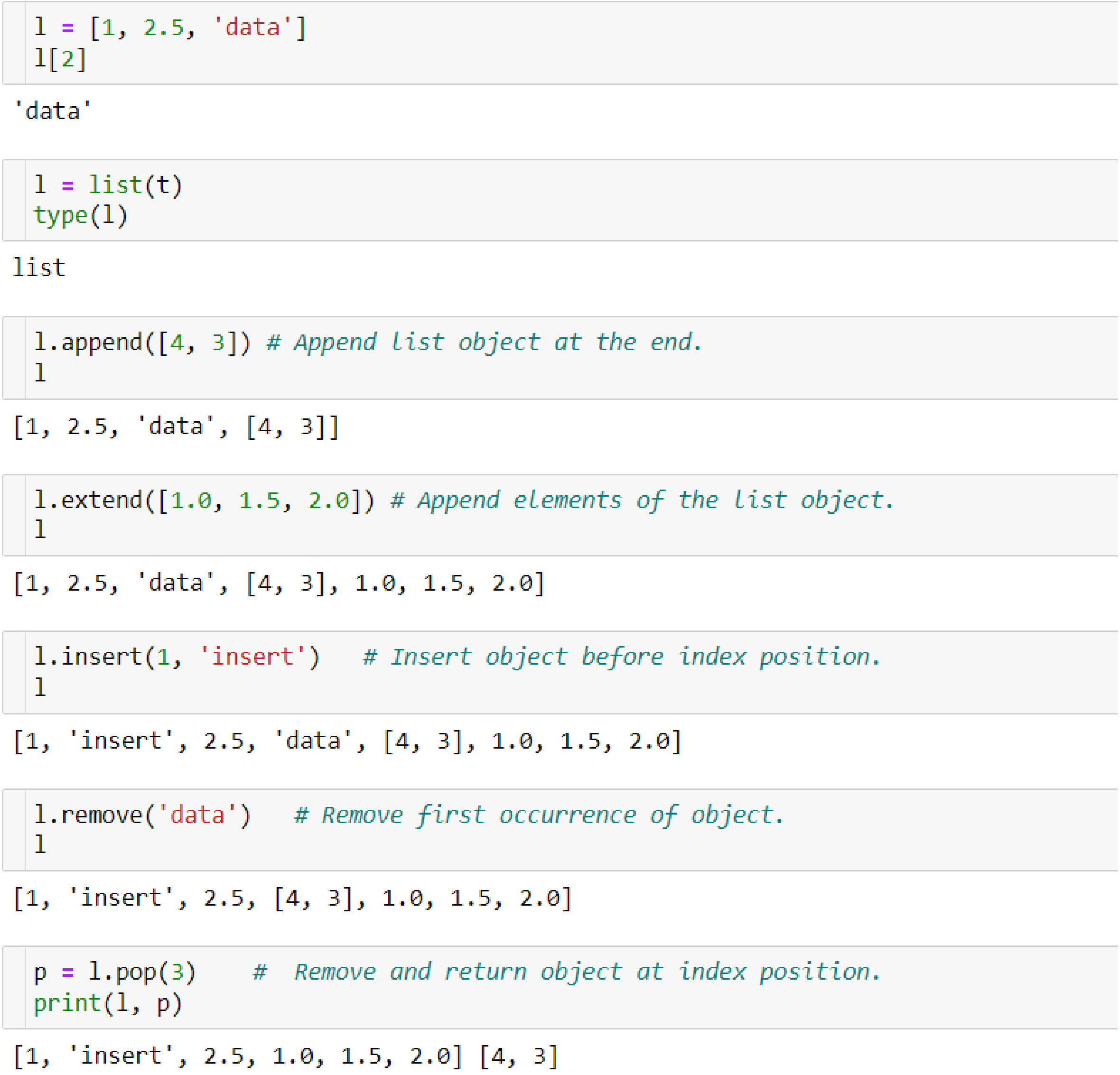
```
1
```

```
t.index('data') # gives the index value of the first appearance of a certain object
```

```
2
```

Tuples are simple and are defined by providing objects in parentheses but are limited in their applications.

Basic Data Structures: *List*



```
l = [1, 2.5, 'data']
l[2]
```

```
'data'
```

```
l = list(t)
type(l)
```

```
list
```

```
l.append([4, 3]) # Append list object at the end.
l
```

```
[1, 2.5, 'data', [4, 3]]
```

```
l.extend([1.0, 1.5, 2.0]) # Append elements of the list object.
l
```

```
[1, 2.5, 'data', [4, 3], 1.0, 1.5, 2.0]
```

```
l.insert(1, 'insert')    # Insert object before index position.
l
```

```
[1, 'insert', 2.5, 'data', [4, 3], 1.0, 1.5, 2.0]
```

```
l.remove('data')    # Remove first occurrence of object.
l
```

```
[1, 'insert', 2.5, [4, 3], 1.0, 1.5, 2.0]
```

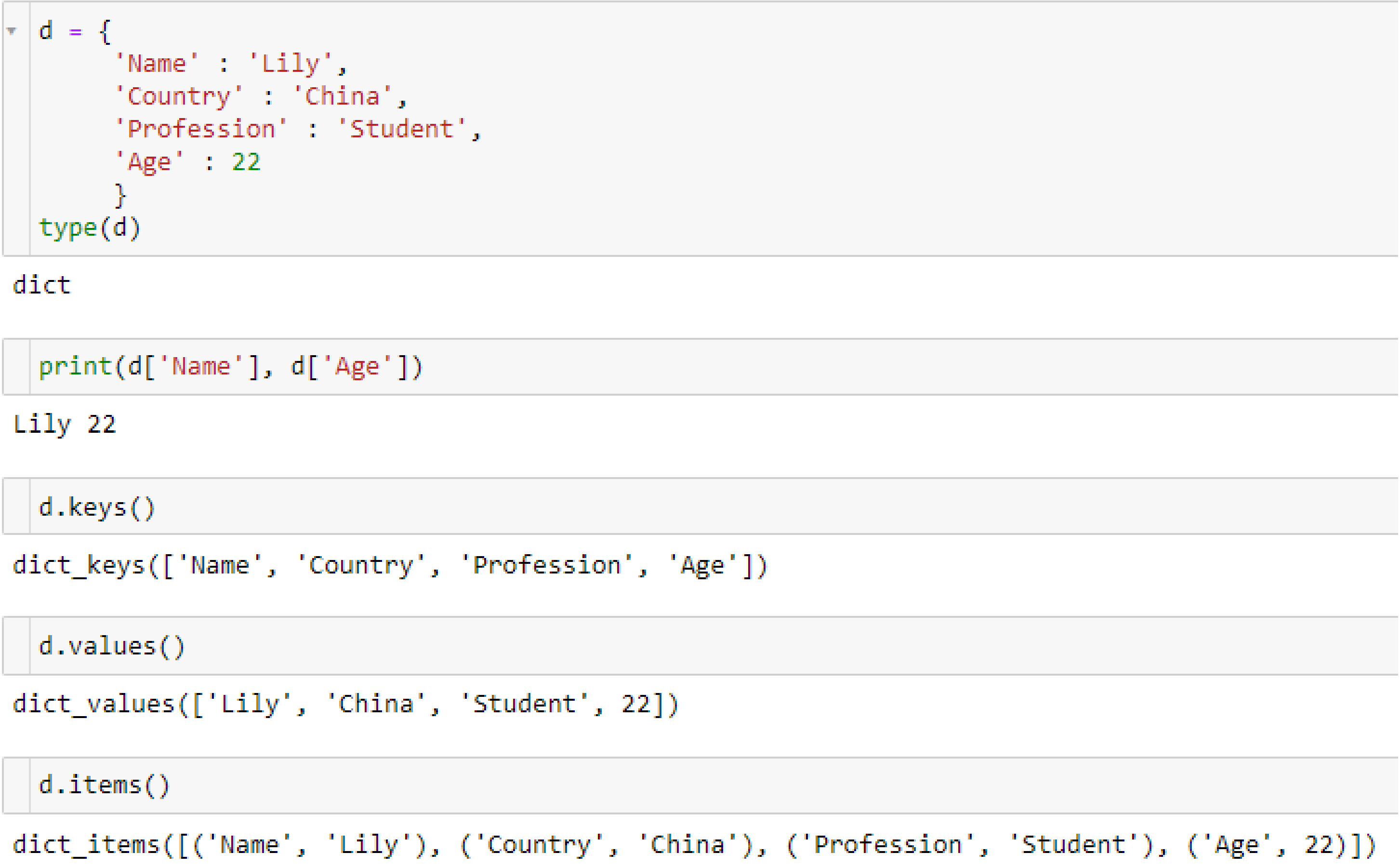
```
p = l.pop(3)    #  Remove and return object at index position.
print(l, p)
```

```
[1, 'insert', 2.5, 1.0, 1.5, 2.0] [4, 3]
```

Lists objects are defined through brackets and the basic capabilities and behaviors are similar to those of tuple objects but is more flexible and powerful.

- List objects are also expandable and reducible via different methods.

Basic Data Structures: *Dictionary*



```
d = {
      'Name' : 'Lily',
      'Country' : 'China',
      'Profession' : 'Student',
      'Age' : 22
      }
type(d)
```

```
dict
```

```
print(d['Name'], d['Age'])
```

```
Lily 22
```

```
d.keys()
```

```
dict_keys(['Name', 'Country', 'Profession', 'Age'])
```

```
d.values()
```

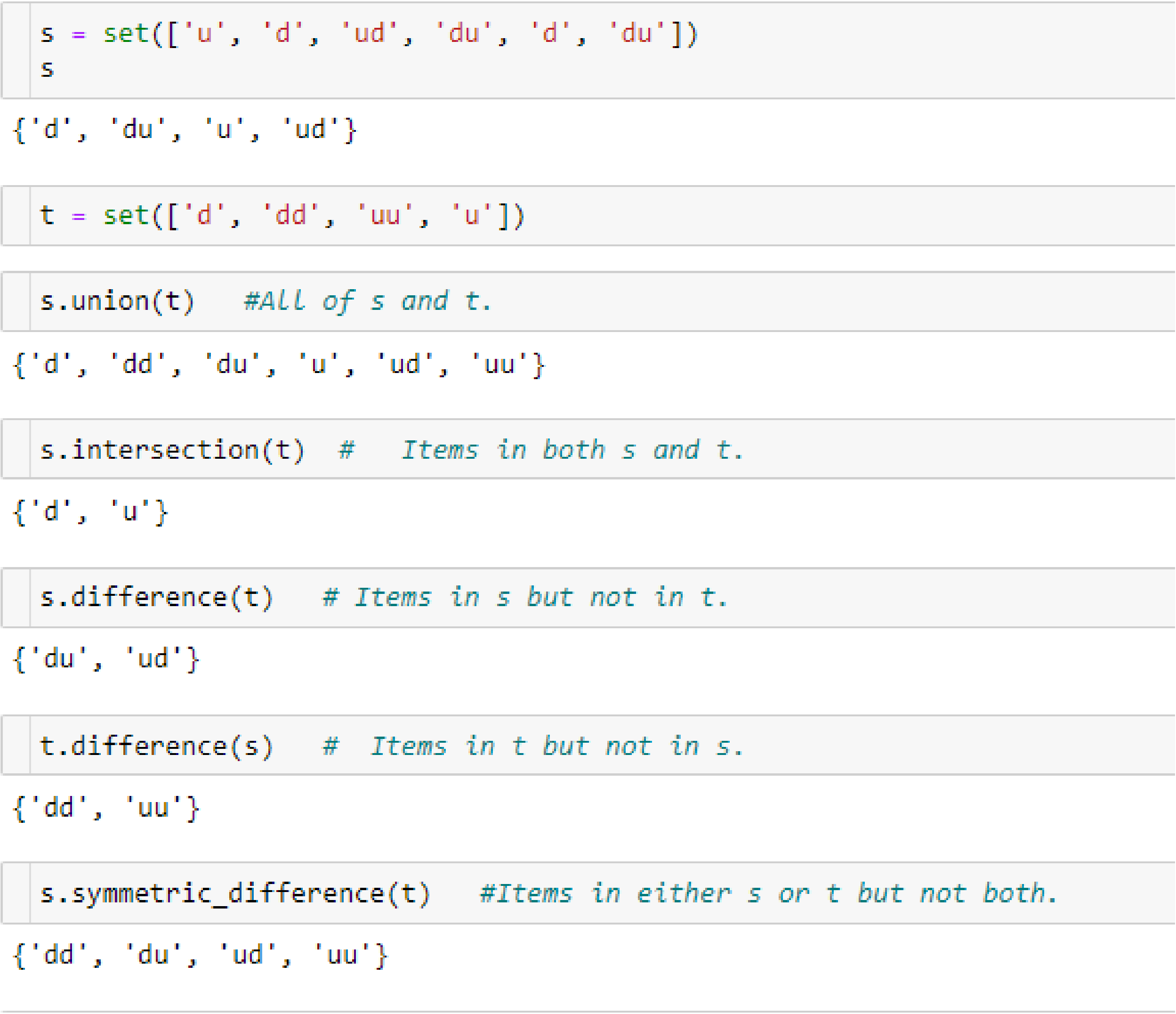
```
dict_values(['Lily', 'China', 'Student', 22])
```

```
d.items()
```

```
dict_items([('Name', 'Lily'), ('Country', 'China'), ('Profession', 'Student'), ('Age', 22)])
```

Dicts are so-called *key-value stores*.
While list objects are ordered and sortable, dictionary objects are unordered and not sortable

Basic Data Structures: *Set*



```
s = set(['u', 'd', 'ud', 'du', 'd', 'du'])
s
```

```
{'d', 'du', 'u', 'ud'}
```

```
t = set(['d', 'dd', 'uu', 'u'])
```

```
s.union(t)   #All of s and t.
```

```
{'d', 'dd', 'du', 'u', 'ud', 'uu'}
```

```
s.intersection(t)  #   Items in both s and t.
```

```
{'d', 'u'}
```

```
s.difference(t)   # Items in s but not in t.
```

```
{'du', 'ud'}
```

```
t.difference(s)   #  Items in t but not in s.
```

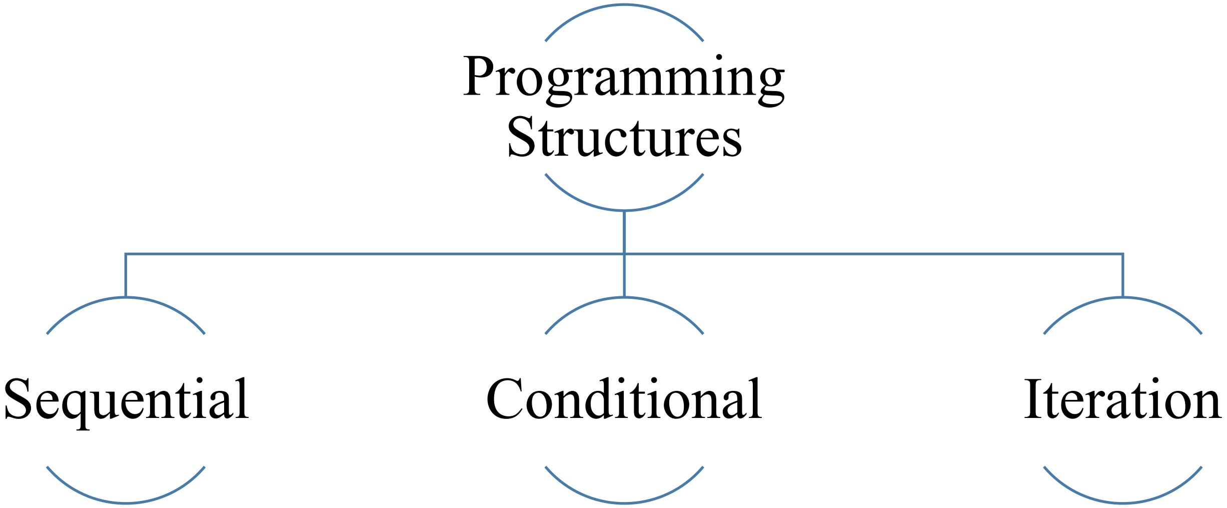
```
{'dd', 'uu'}
```

```
s.symmetric_difference(t)   #Items in either s or t but not both.
```

```
{'dd', 'du', 'ud', 'uu'}
```

Sets are unordered collections of other objects, containing every element only once

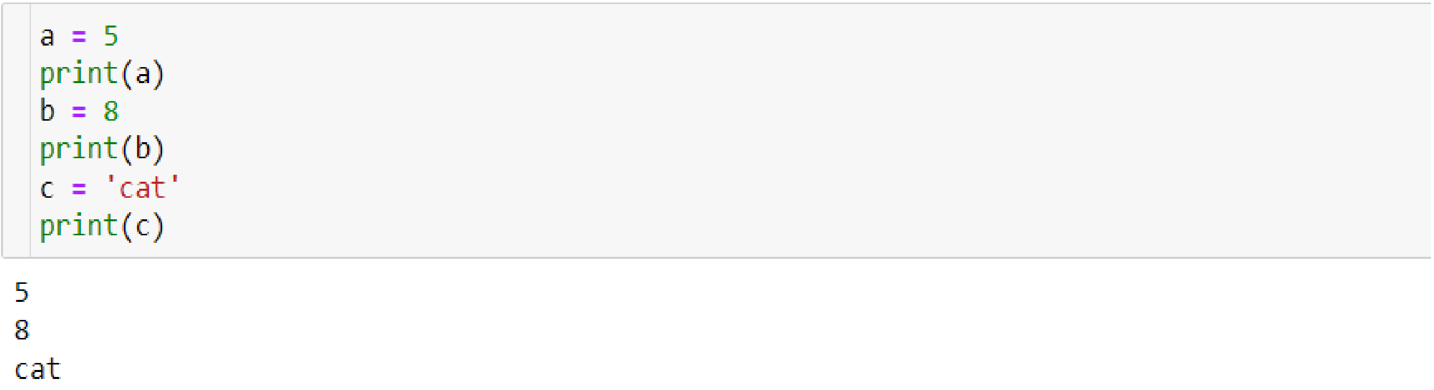
Programming Structures



Programming Structures: *sequential*

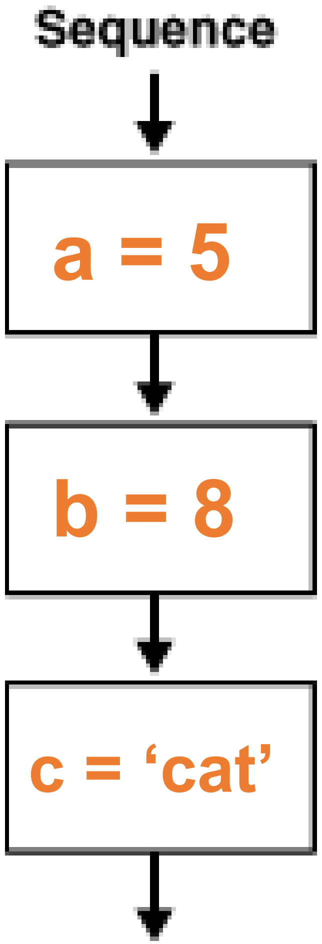
SEQUETIAL

- Programs are mostly written sequentially, meaning the first line of program runs first followed by the program in the second line, then the third line and so on.



```
a = 5
print(a)
b = 8
print(b)
c = 'cat'
print(c)
```

```
5
8
cat
```



First line executes first!

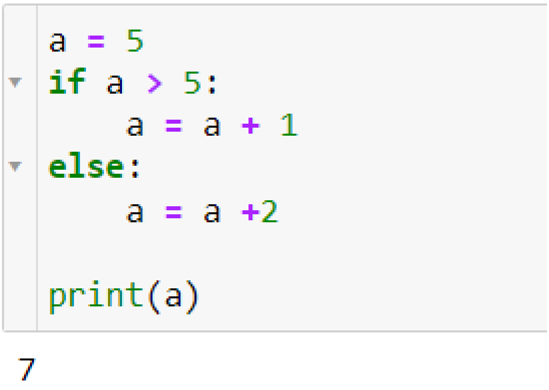
second

third and so on…

Programming Structures: *Conditional*

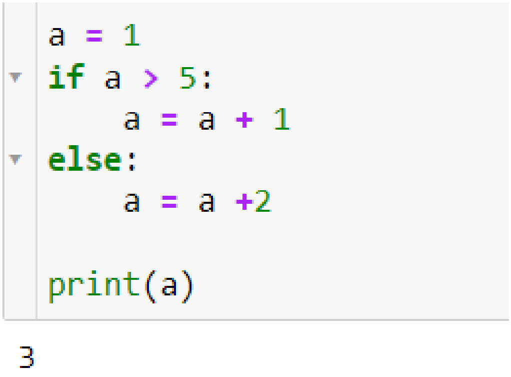
CONDITIONAL

- Programs become more useful when we can change its behavior given a condition is satisfied.



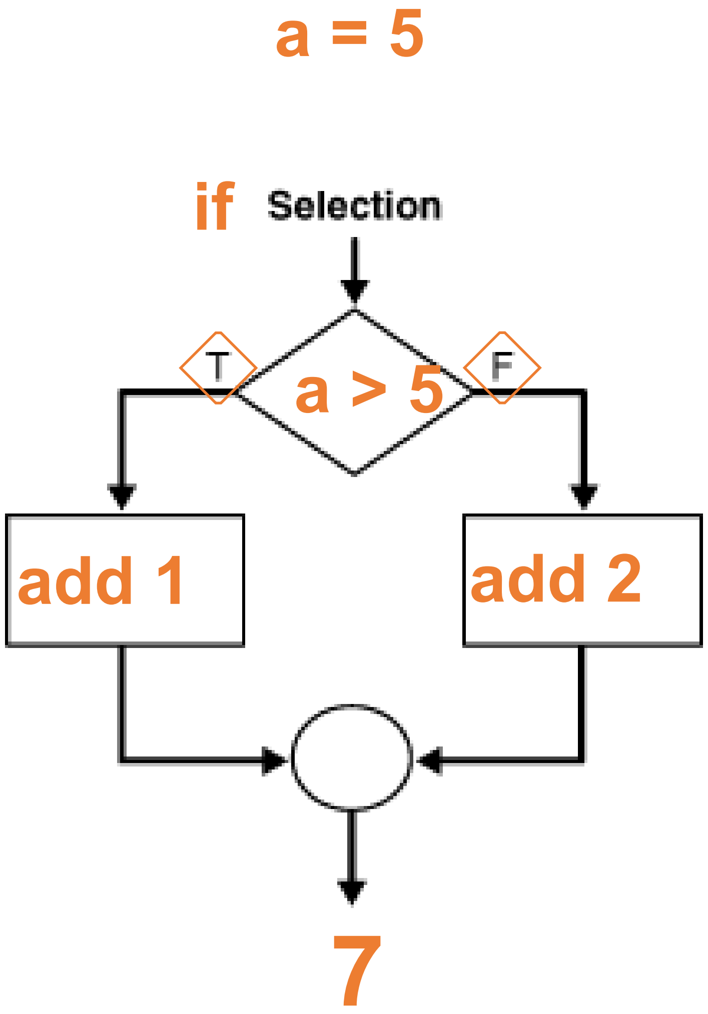
```
a = 5
if a > 5:
    a = a + 1
else:
    a = a +2

print(a)
```

```
7
```



```
a = 1
if a > 5:
    a = a + 1
else:
    a = a +2

print(a)
```

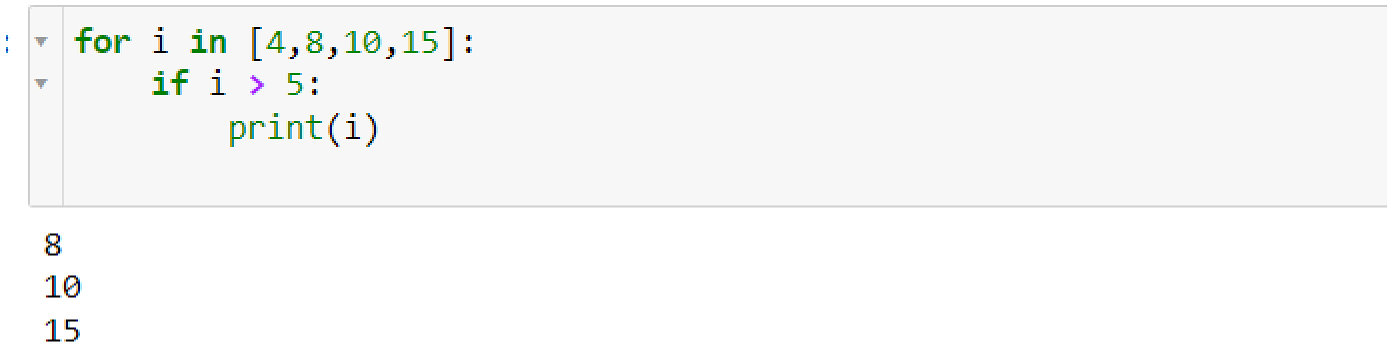
```
3
```



Programming Structures: *Iteration*

ITERATION

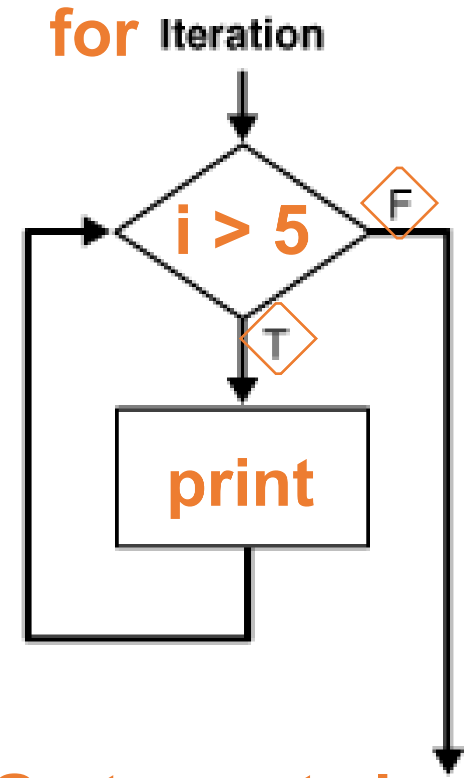
- Programs become powerful when the same block of code can be repeatedly executed on either identical tasks or similar tasks.



```
for i in [4,8,10,15]:
    if i > 5:
        print(i)
```

```
8
10
15
```

numlist = [4,8,10,15]



[8,10,15] ← Go to next element and repeat until end

Programming Structures: *Iteration with Conditional*

```
numlist = [4,8,10,15]
```

1. What if we want to add 1 to each item of this list if greater than 5?

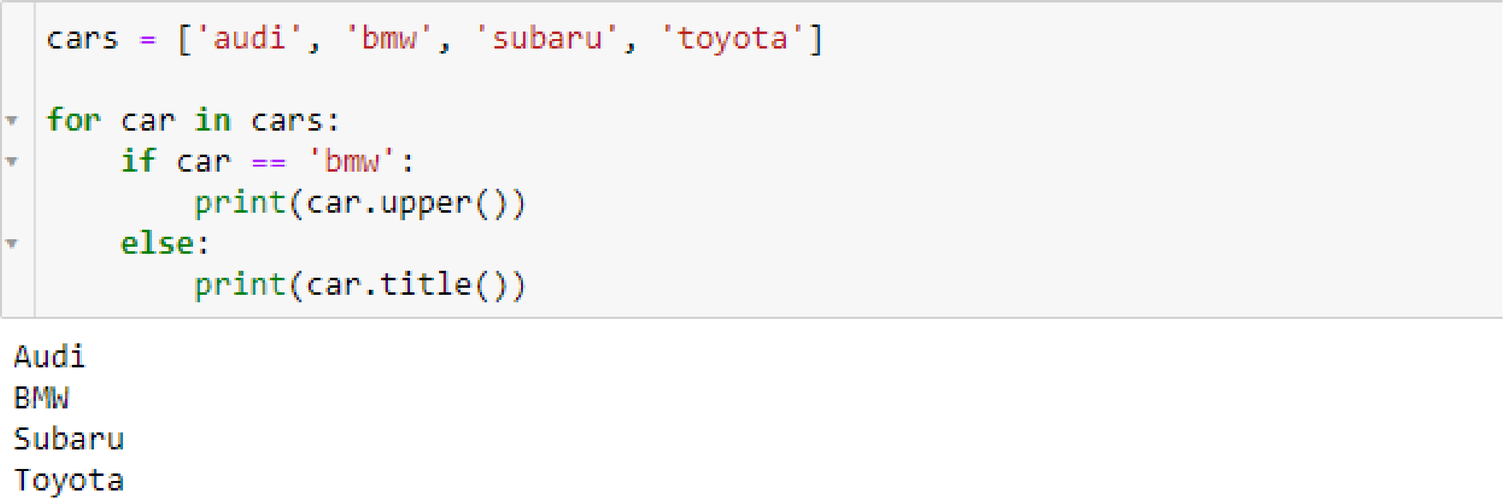
→ Use conditional and iteration together

```
for i in numlist:
    if i > 5:
        print(i+1)
    else:
        print(i)
```

```
→ 4
  9
  11
  16
```

Examples: For loop + if statement

Imagine you have a list of cars and you want to print out the name of each car. Car names are proper names, so the names of most cars should be printed in title case. However, the value 'bmw' should be printed in all uppercase. The following code loops through a list of car names and looks for the value 'bmw'. Whenever the value is 'bmw', it’s printed in uppercase instead of title case:

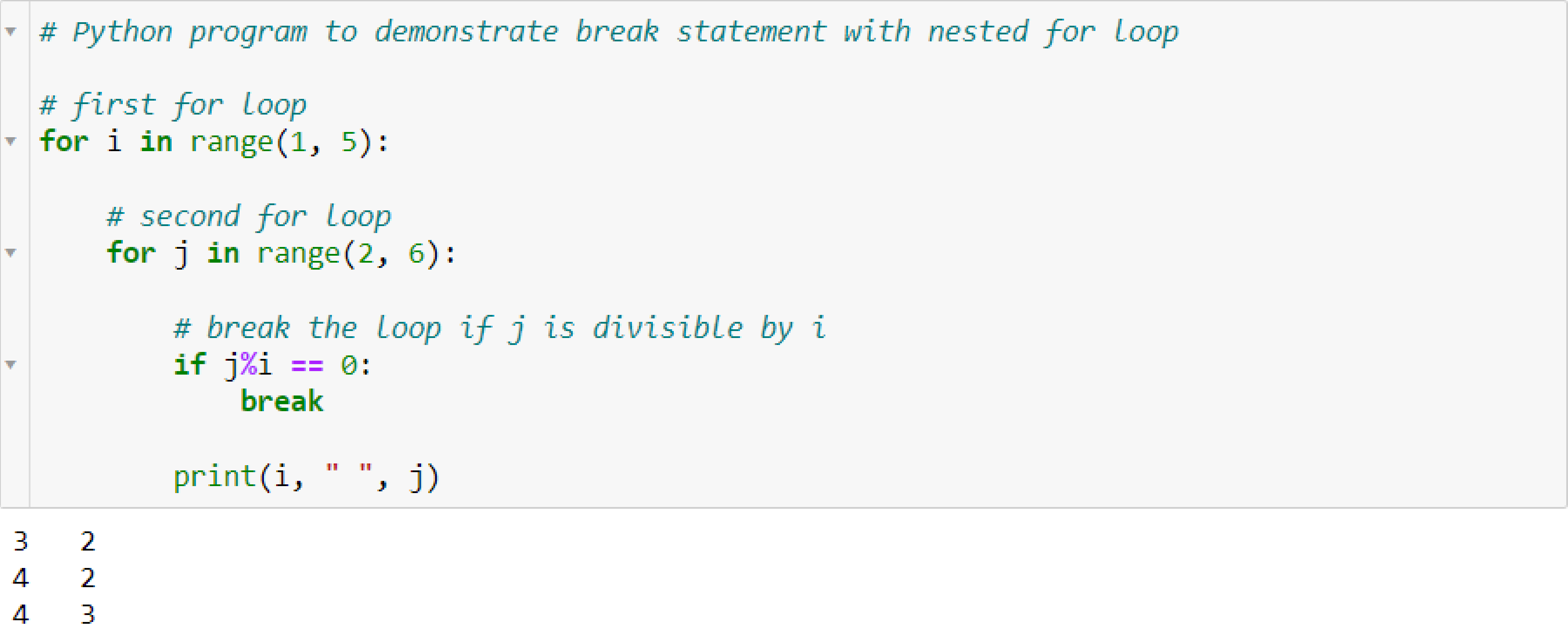


```python
cars = ['audi', 'bmw', 'subaru', 'toyota']

for car in cars:
    if car == 'bmw':
        print(car.upper())
    else:
        print(car.title())
```

```
Audi
BMW
Subaru
Toyota
```

Examples: Pass, continue and break statements in loops

- Break: Terminate the loop once a specified condition is met.

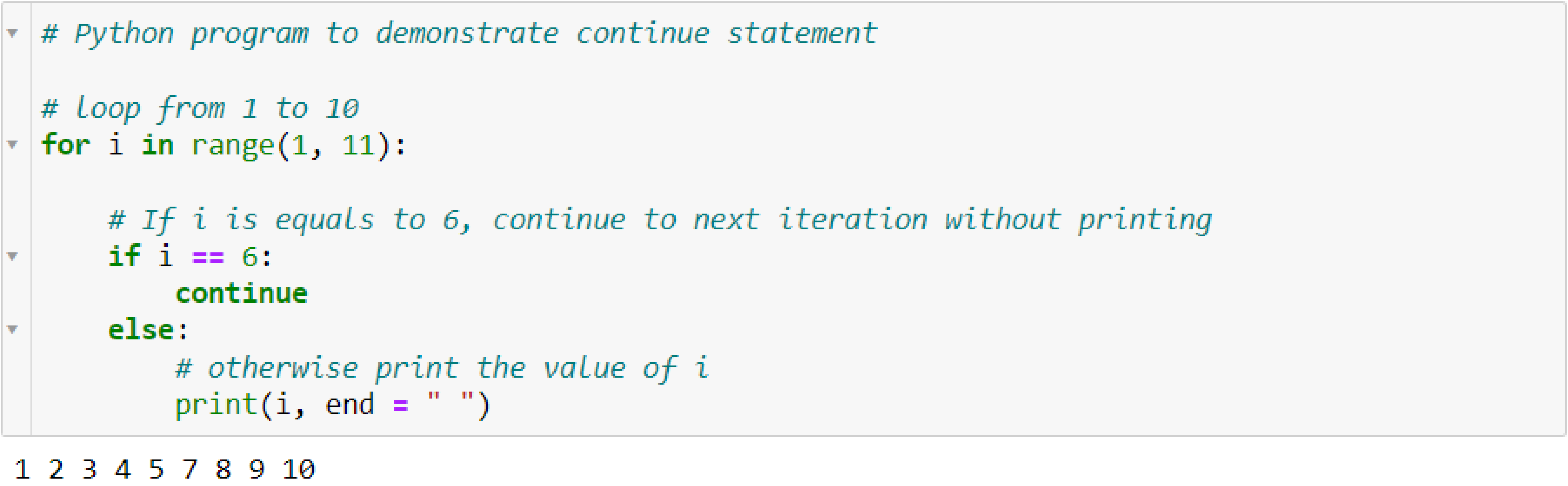


```python
# Python program to demonstrate break statement with nested for loop

# first for loop
for i in range(1, 5):

    # second for loop
    for j in range(2, 6):

        # break the loop if j is divisible by i
        if j%i == 0:
            break

        print(i, " ", j)
```

```
3   2
4   2
4   3
```

Examples: Pass, continue and break statements in loops

- Continue: Skip the remaining code inside a loop for the current iteration only.

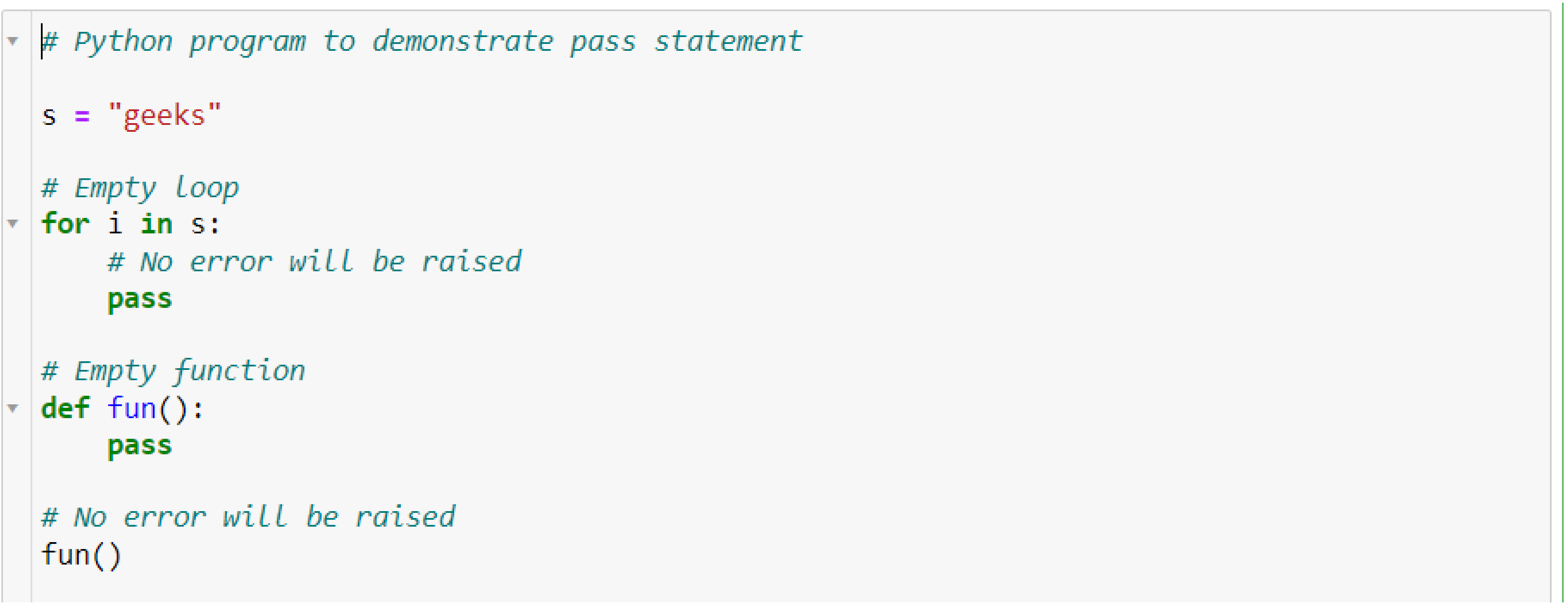


```python
# Python program to demonstrate continue statement

# loop from 1 to 10
for i in range(1, 11):

    # If i is equals to 6, continue to next iteration without printing
    if i == 6:
        continue
    else:
        # otherwise print the value of i
        print(i, end = " ")
```

```
1 2 3 4 5 7 8 9 10
```

Examples: Pass, continue and break statements in loops

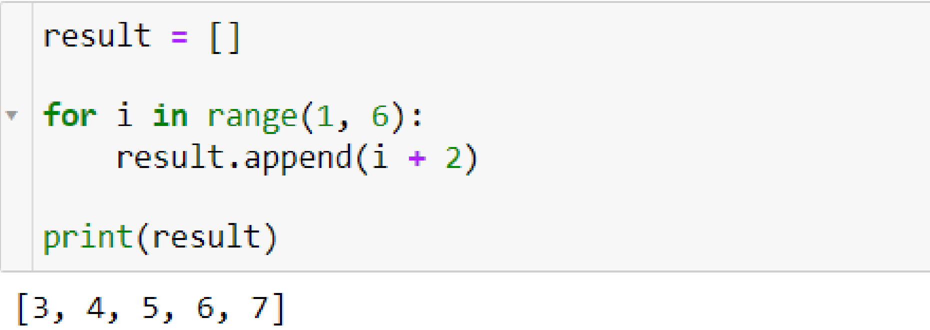
- Pass: Simple do nothing. It’s typically used as a placeholder for future code.



```python
# Python program to demonstrate pass statement

s = "geeks"

# Empty loop
for i in s:
    # No error will be raised
    pass

# Empty function
def fun():
    pass

# No error will be raised
fun()
```

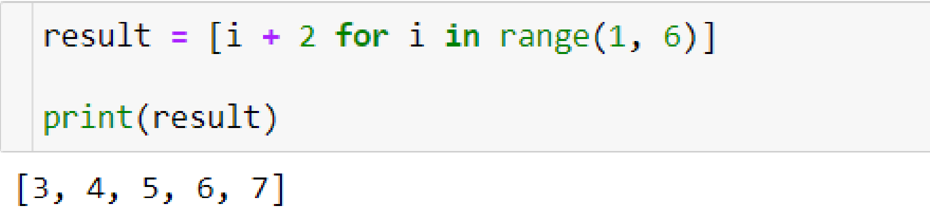
Examples: One line for loop

- One line for loop based on list makes the codes shorter



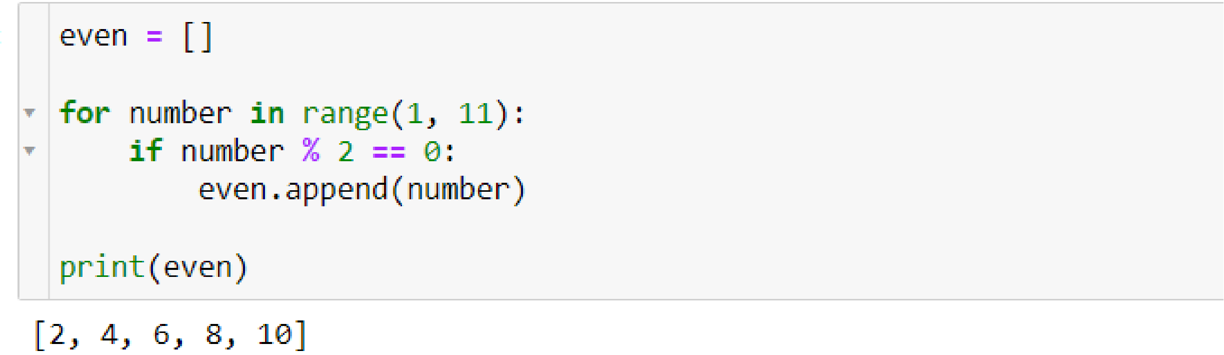
```
result = []

for i in range(1, 6):
    result.append(i + 2)

print(result)
```

```
[3, 4, 5, 6, 7]
```



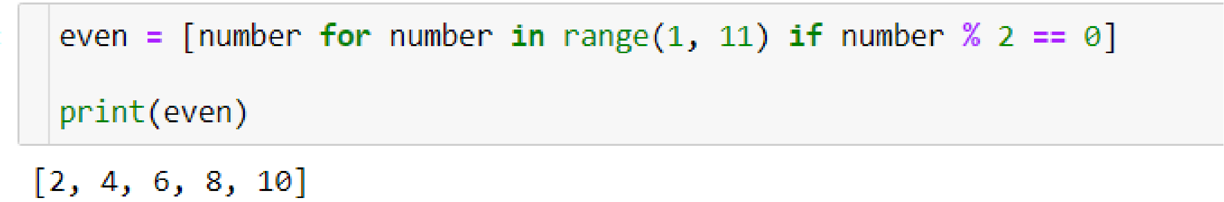
```
result = [i + 2 for i in range(1, 6)]

print(result)
```

```
[3, 4, 5, 6, 7]
```



```
even = []

for number in range(1, 11):
    if number % 2 == 0:
        even.append(number)

print(even)
```

```
[2, 4, 6, 8, 10]
```

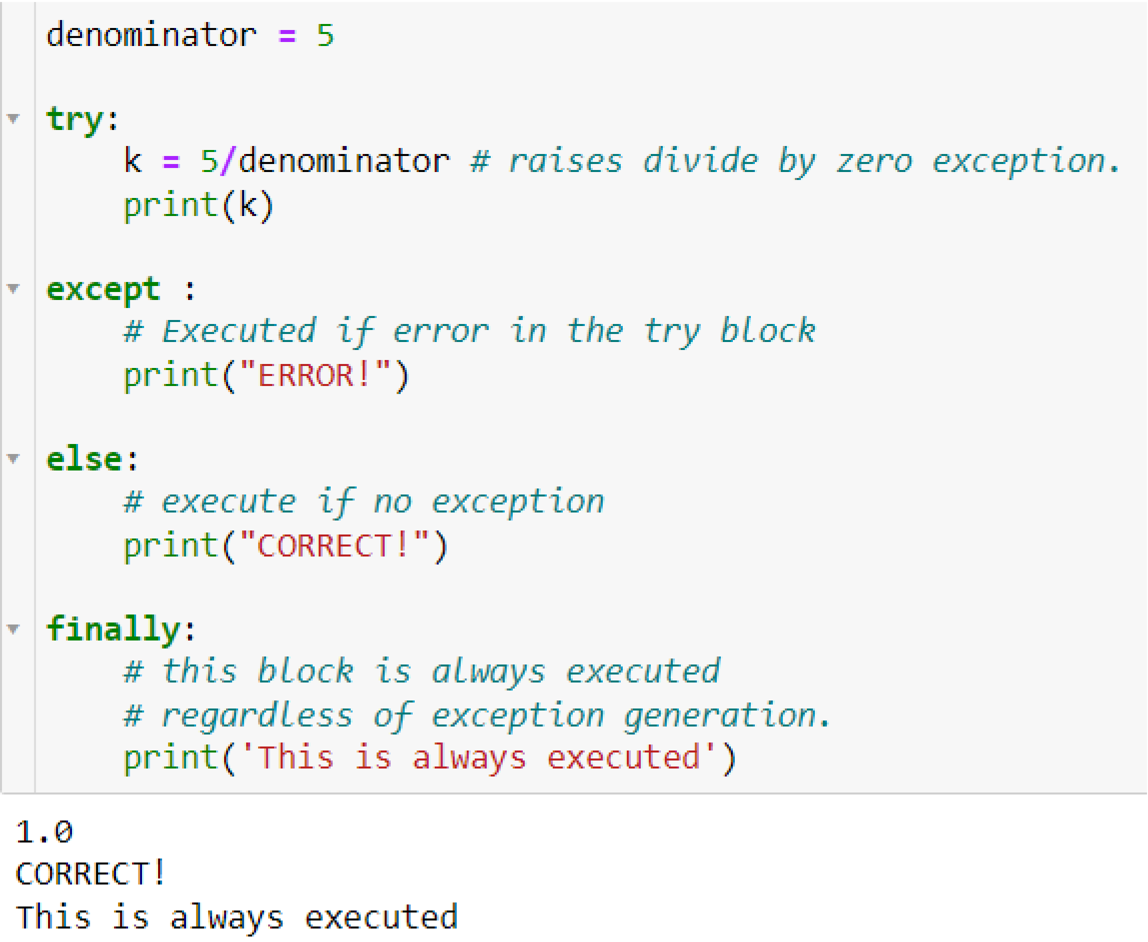


```
even = [number for number in range(1, 11) if number % 2 == 0]

print(even)
```

```
[2, 4, 6, 8, 10]
```

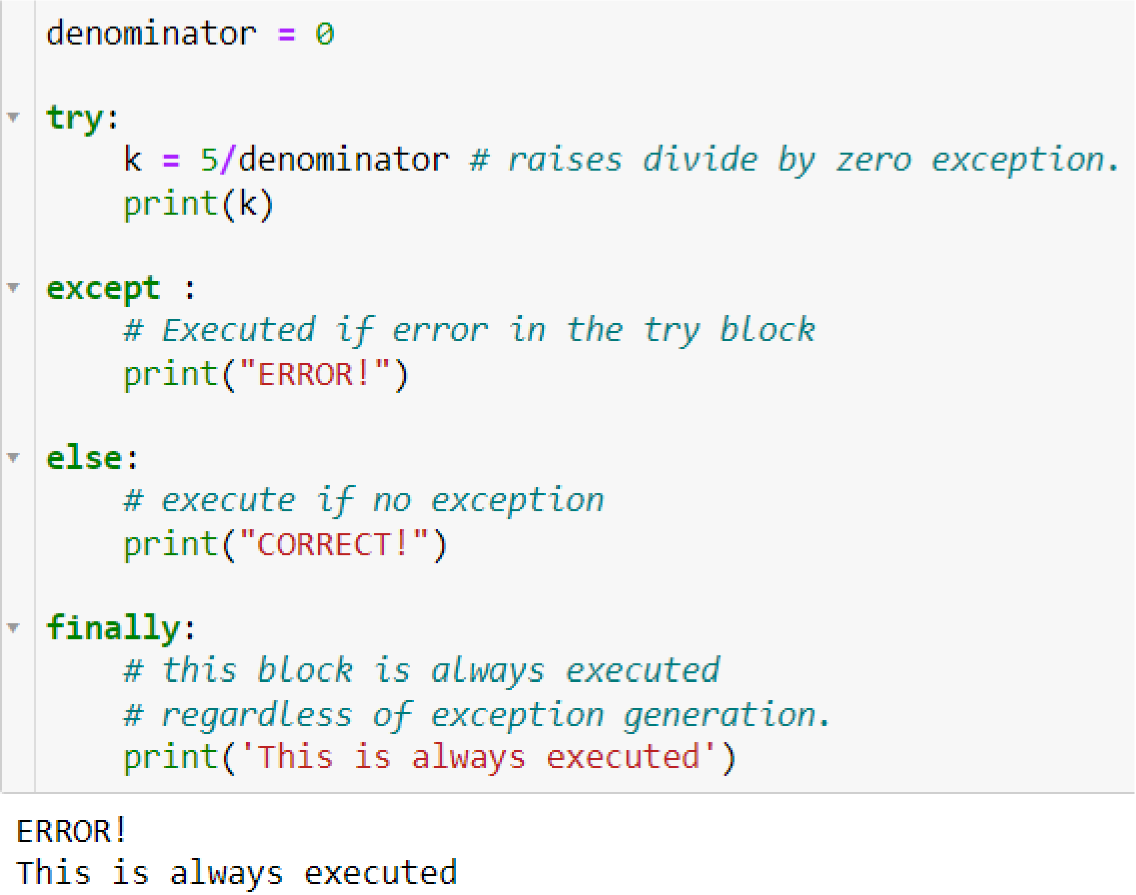
Examples: Try… Except…

- Using except to execute codes with errors



```
denominator = 5

try:
    k = 5/denominator # raises divide by zero exception.
    print(k)

except :
    # Executed if error in the try block
    print("ERROR!")

else:
    # execute if no exception
    print("CORRECT!")

finally:
    # this block is always executed
    # regardless of exception generation.
    print('This is always executed')
```

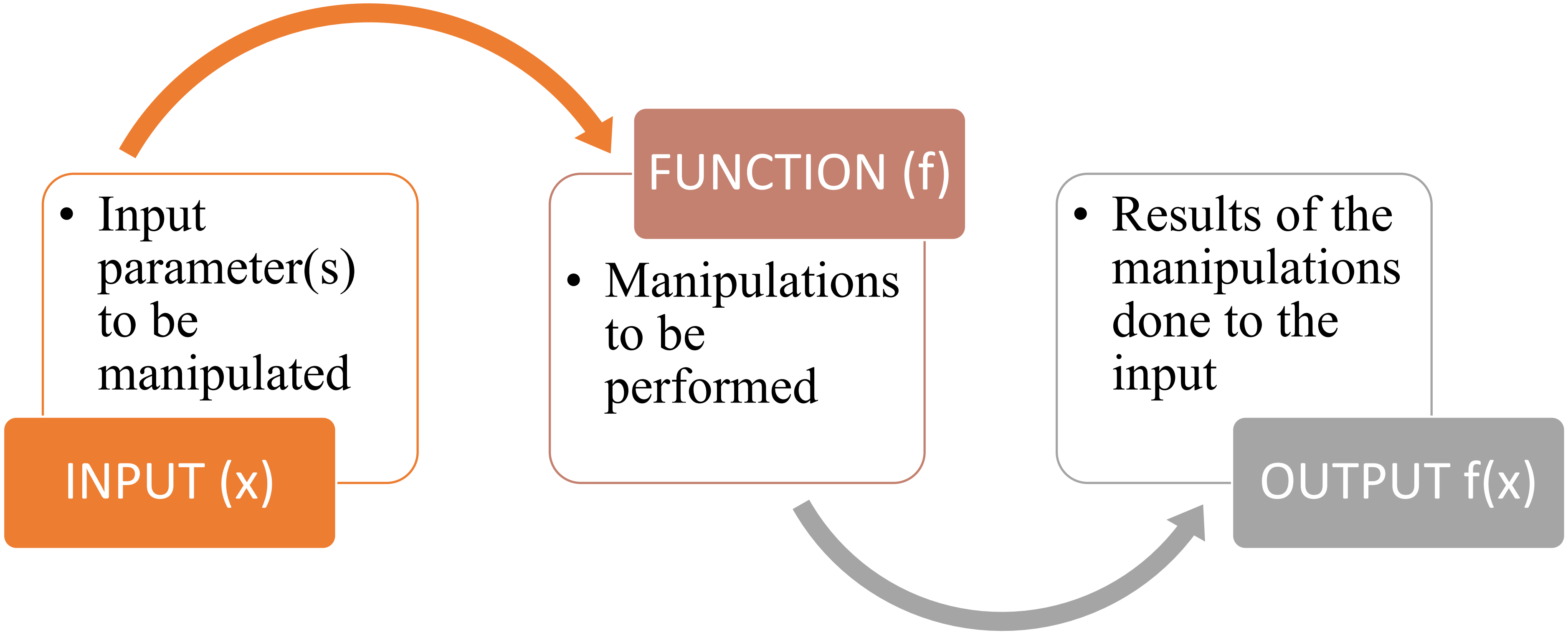
```
1.0
CORRECT!
This is always executed
```



```
denominator = 0

try:
    k = 5/denominator # raises divide by zero exception.
    print(k)

except :
    # Executed if error in the try block
    print("ERROR!")

else:
    # execute if no exception
    print("CORRECT!")

finally:
    # this block is always executed
    # regardless of exception generation.
    print('This is always executed')
```

```
ERROR!
This is always executed
```

Functions



INPUT (x)

- Input parameter(s) to be manipulated

FUNCTION (f)

- Manipulations to be performed

OUTPUT f(x)

- Results of the manipulations done to the input

User-Defined Functions

```
numlist = [4,8,10,15]
```

```
for i in nextlist:
    if i > 5:
        print(i+1)
    else:
        print(i)
```

1. What if we want to add 1 to each item of this list if greater than 5?

→ Conditional and iteration!

2. What if we want to be able to do it for many other lists?

→ Copy paste and change the name of list

→ Actually, it is way more efficient to create a function !

User-Defined Functions (Contd.)

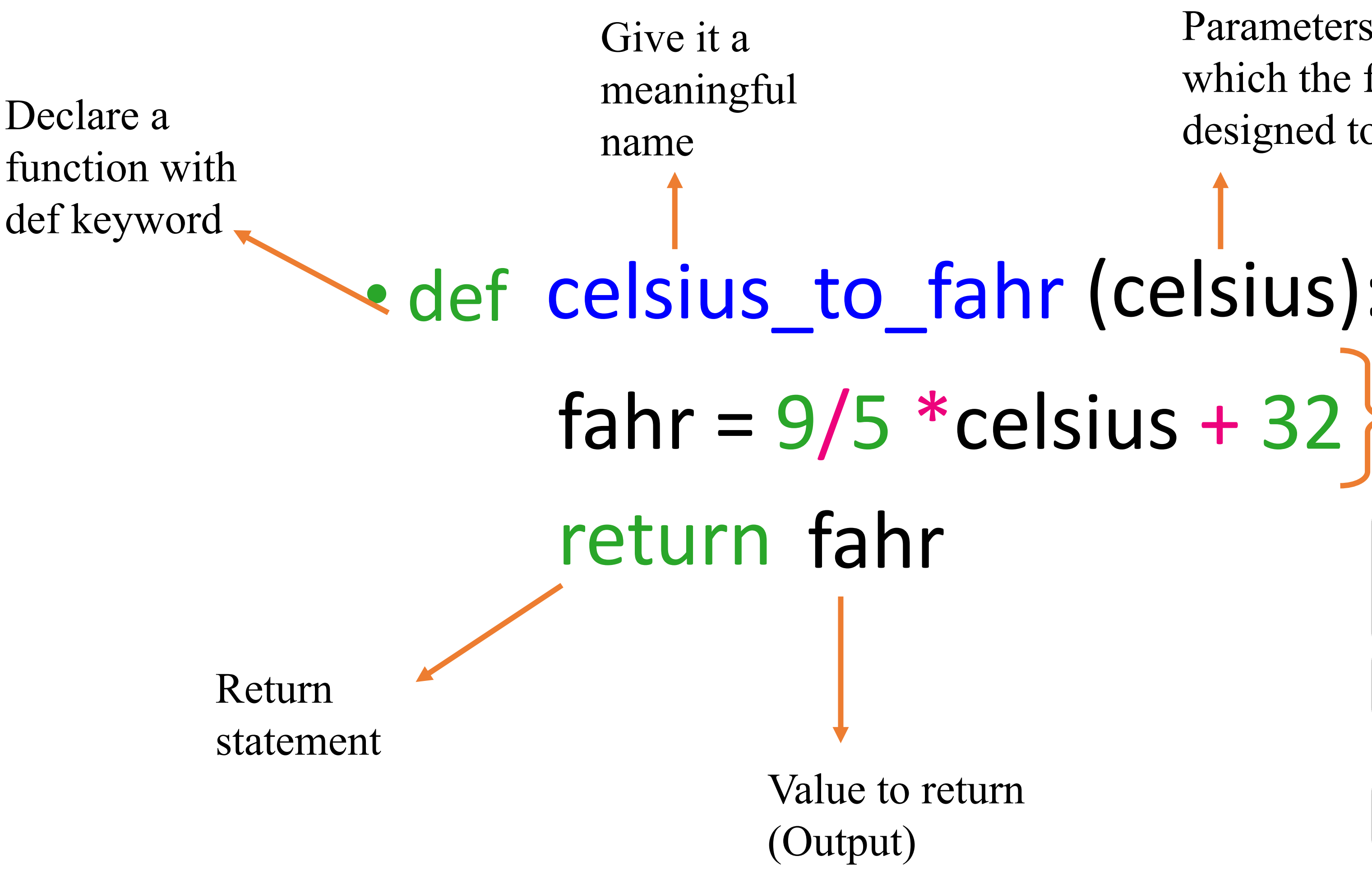
```
def fiveormore(alist):
    for i in alist:
        if i > 5:
            print(i+1)
        else:
            print(i)
```

```
numlist = [4,8,10,15]
fiveormore(numlist)
→ 4
  9
  11
  16
```

```
nextlist = [1,3,4]
fiveormore(nextlist)
→ 1
  3
  4
```

```
alist = [9,11,15]
fiveormore(alist)
→ 10
  12
  16
```

User-Defined Functions (Cont.)



Give it a meaningful name

Parameters or input on which the function is designed to work

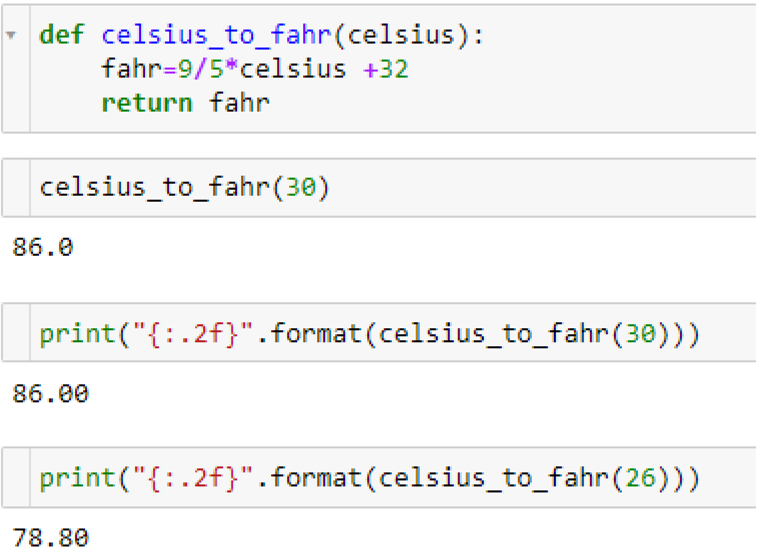
Declare a function with def keyword

```
• def  celsius_to_fahr (celsius):
        fahr = 9/5 *celsius + 32
        return  fahr
```

Block of code that defines the operation to run

Return statement

Value to return (Output)



```
def celsius_to_fahr(celsius):
    fahr=9/5*celsius +32
    return fahr
```

```
celsius_to_fahr(30)
```

```
86.0
```

```
print("{:.2f}".format(celsius_to_fahr(30)))
```

```
86.00
```

```
print("{:.2f}".format(celsius_to_fahr(26)))
```

```
78.80
```

This Week

- Installation: Anaconda and jupyter notebook

- Basic Data Types and Structures:
  - Integer, Float, Bool, String
  - Tuple, List, Dictionary, Set

- Programming Structures:
  - Sequential, Conditional, Iteration
  - Examples

- User-Defined Functions

Next Week

- Object, class, method
- Read and write file
- Numpy
- Pandas
- …