# Python

# Agenda

- Introduction
- Why Python
- IDE - Why Jupyter for Python/Anaconda for Data Science
- Getting started with Python
- Data Structures
- Conditional Statements
- Loops
- Control Statements
- List Comprehensions
- Functions

## Why Python ?

## Who gifted Python ?

## Guido Van Rossam

- 1989
- National Research Institute in Netherland

## Guido VanRossam -

*"Over six years ago, in December 1989, I was looking for a hobby programming project that would keep me occupied during the week around Christmas. My office would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood and a big fan of Monty Python's Flying Circus."*

## Features of Python

## Simple & Easy To Learn

```
a=3
b=5
Sum=a+b
```

Open Source

High-level

Interpreted

Large community

**Java**

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world");
    }
}
```

**Python**

```
print("Hello, world")
```

It's that **SIMPLE**!

## Web Development

- Develop web applications
- Scrape websites

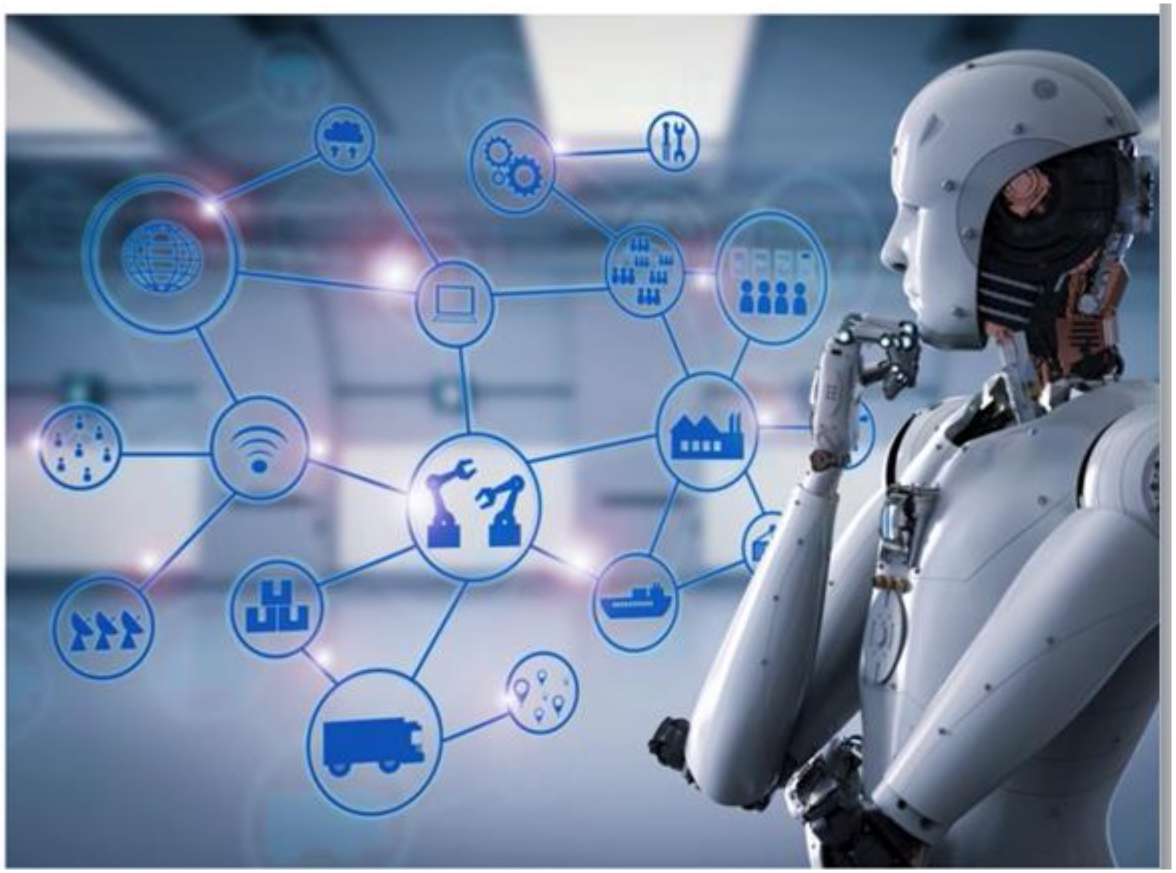
Frameworks

django | Flask | Pylons | WEB2PY

## Artificial Intelligence

**Libraries**

- Scikit-learn
- Keras
- Tensorflow
- Opencv



## Big Data

- Python handles **BIG DATA!**
- Python supports parallel computing
- You can write MapReduce codes in Python

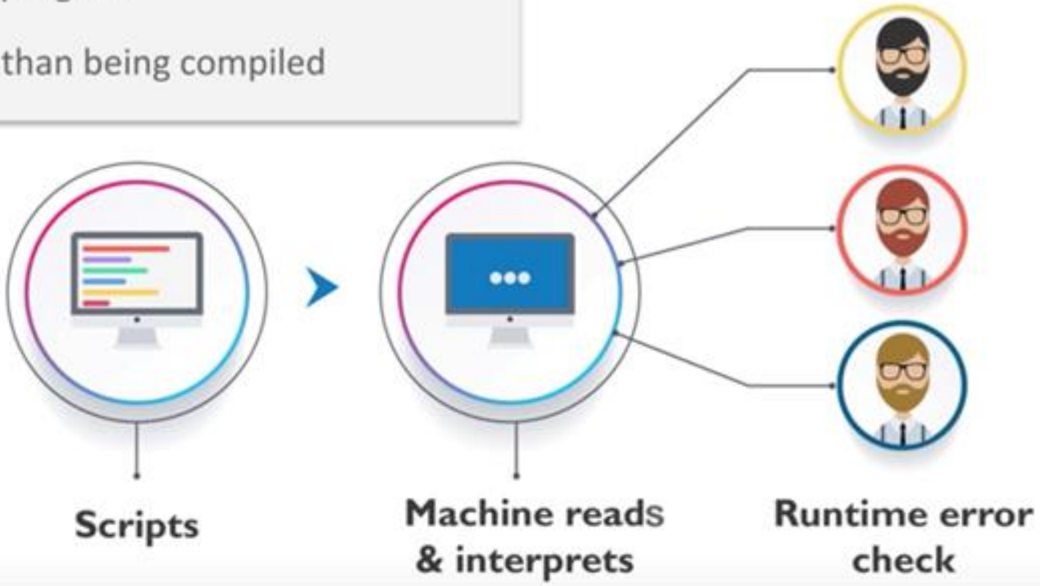
Libraries

PYDOOP | DASK | PySpark

hadoop HDFS

## Scripting: Automation

- It is the most popular scripting language in the industry
- Automate certain tasks in a program
- They are interpreted rather than being compiled



## Data Science

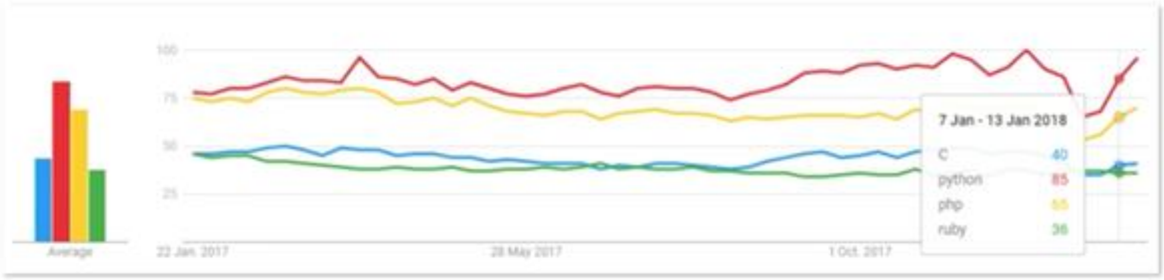
- Well-suited for data manipulation & analysis
- Deals with tabular data with heterogeneously-typed columns
- Arbitrary matrix data
- Observational/ statistical datasets

Libraries

NumPy | Pandas | matplotlib | seaborn

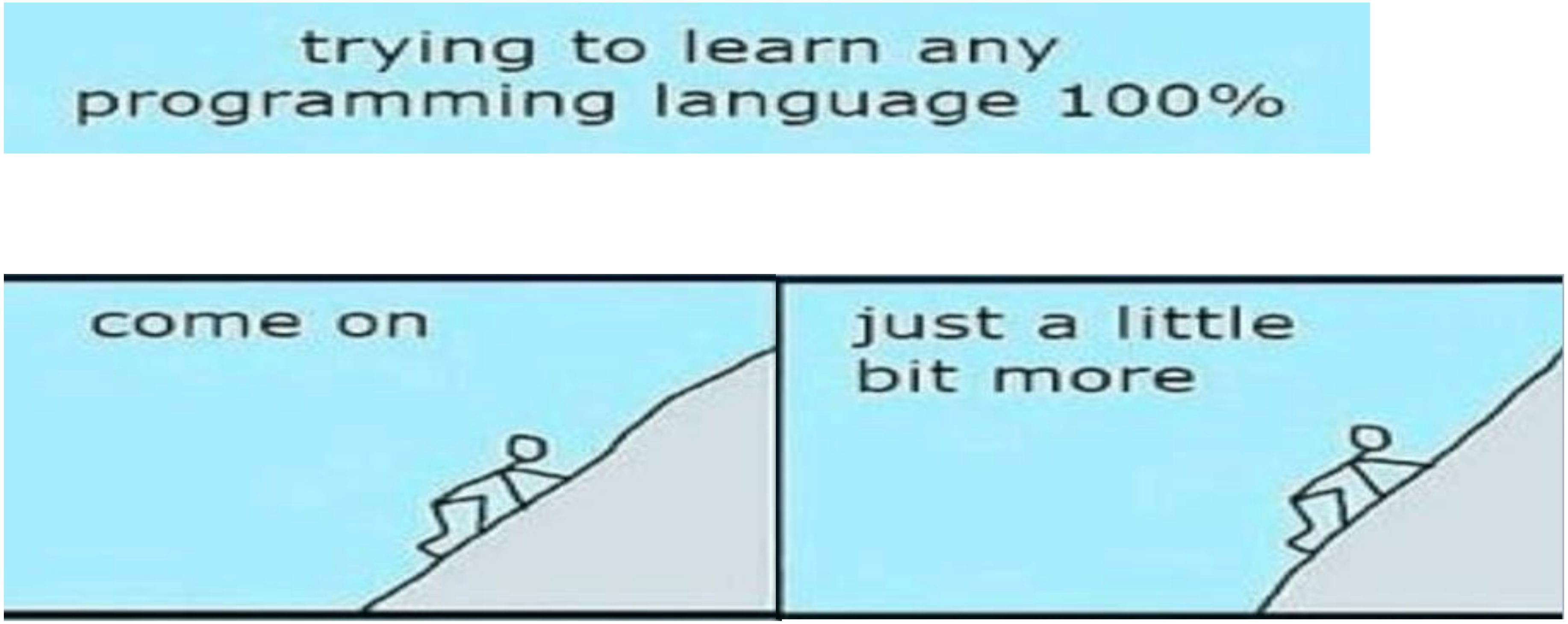
## Popularity & High Salary

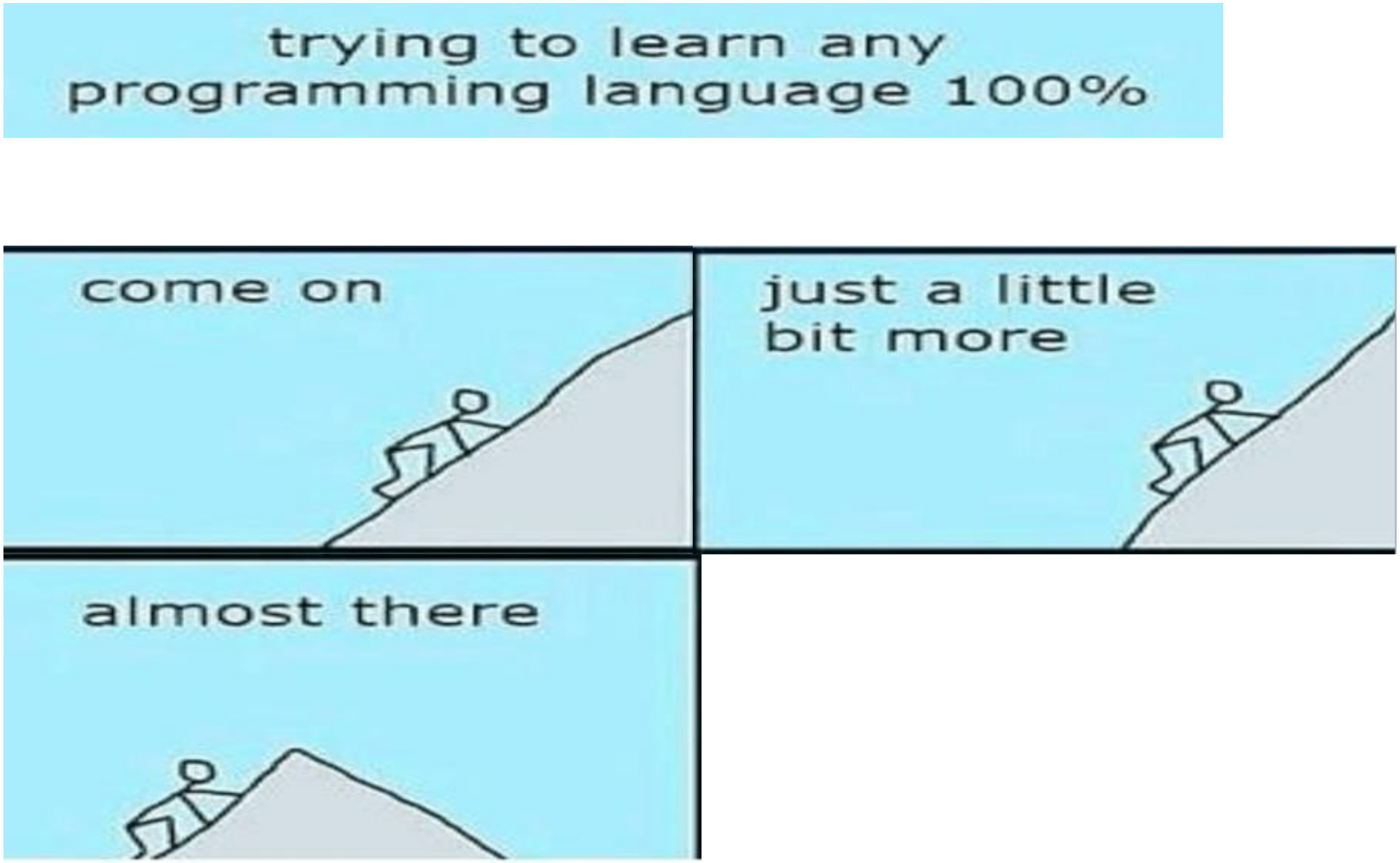
USD $116,028

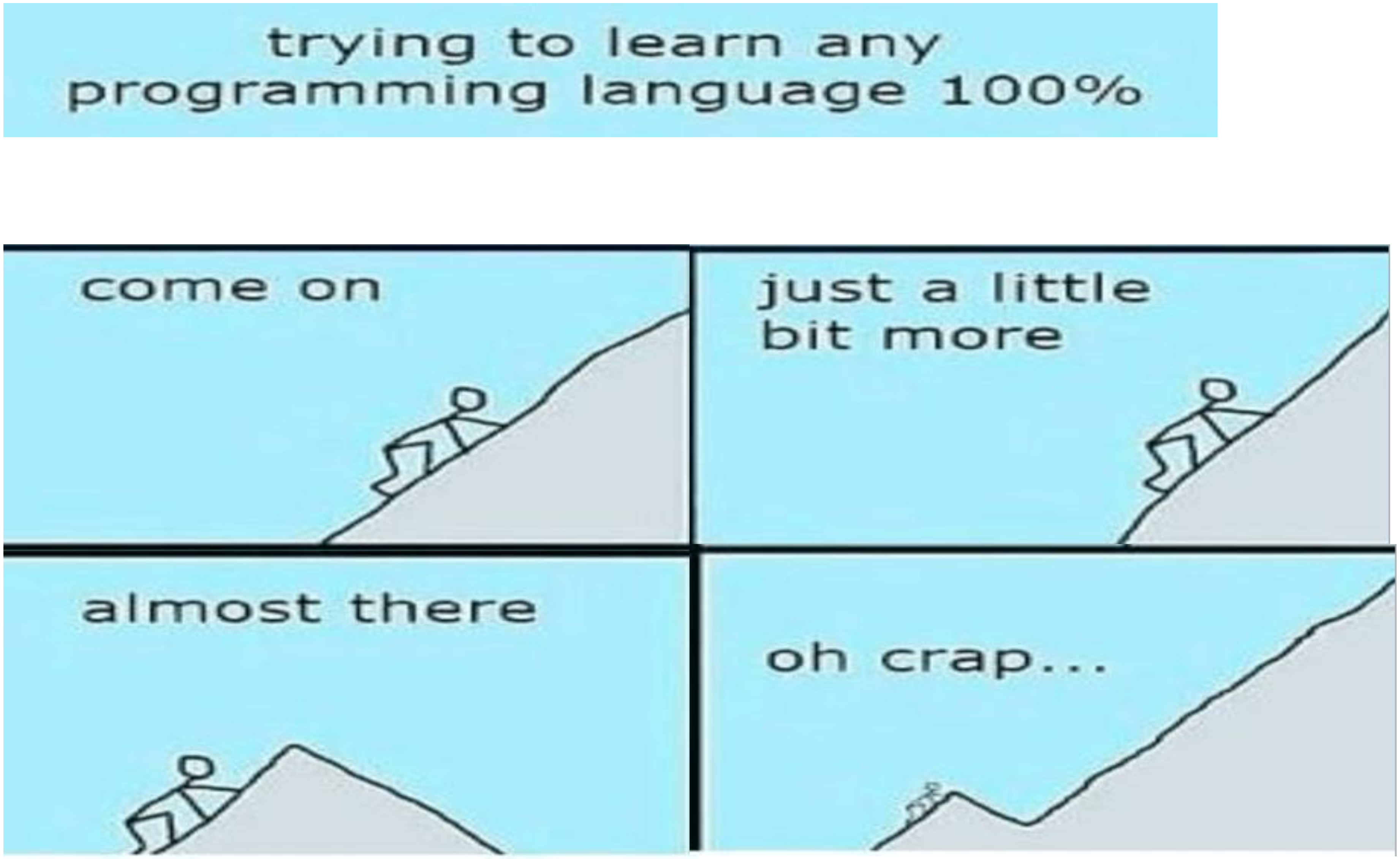


Big Giants

Google YouTube NASA
IBM RaspberryPi facebook. Dropbox


trying to learn any programming language 100%
come on
just a little bit more


trying to learn any programming language 100%
come on
just a little bit more
almost there

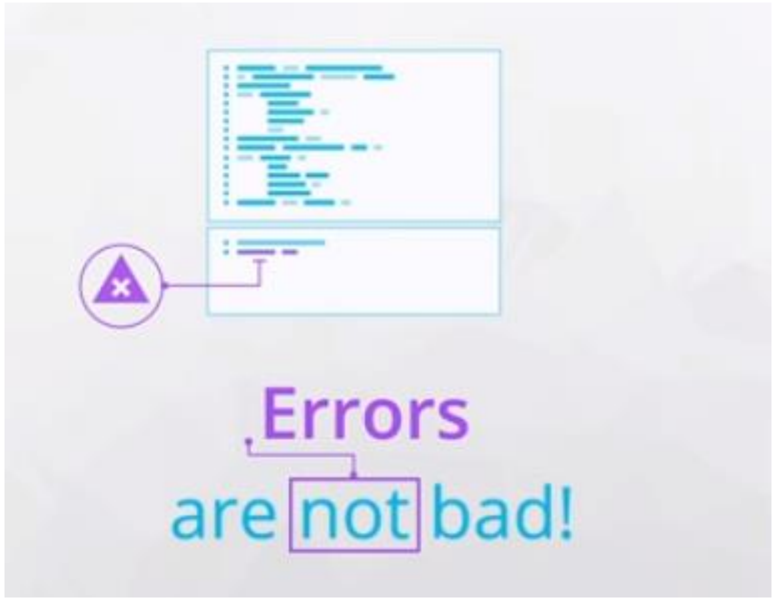

trying to learn any programming language 100%
come on
just a little bit more
almost there
oh crap...

## Lets Get Started with Python

- Launch Anaconda/Jupyter

Aa
case sensitive
Case Sensitive

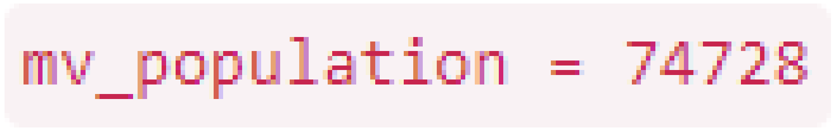
spacing
matters


Errors
are not bad!

## Data Types

1. Integers
2. Floats
3. Boolean
4. String

# Variables & Assignment Operators



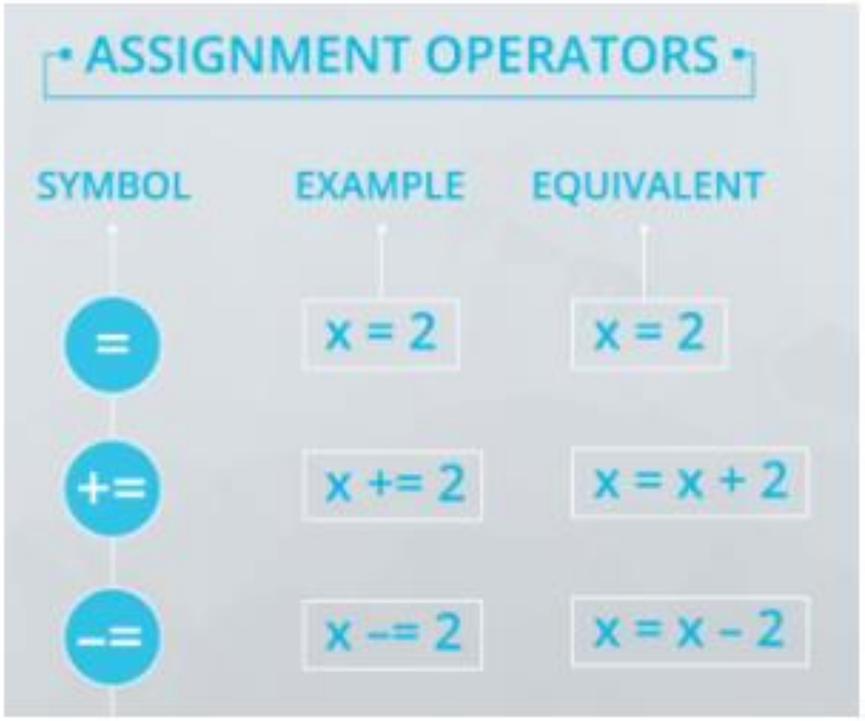
```
mv_population = 74728
```

**NOTE: You can’t use reserved words or built-in identifiers**

Keywords in Python programming language

| | | | | |
|---|---|---|---|---|
| False | class | finally | is | return |
| None | continue | for | lambda | try |
| True | def | from | nonlocal | while |
| and | del | global | not | with |
| as | elif | if | or | yield |
| assert | else | import | pass | |
| break | except | in | raise | |

# Variables & Assignment Operators



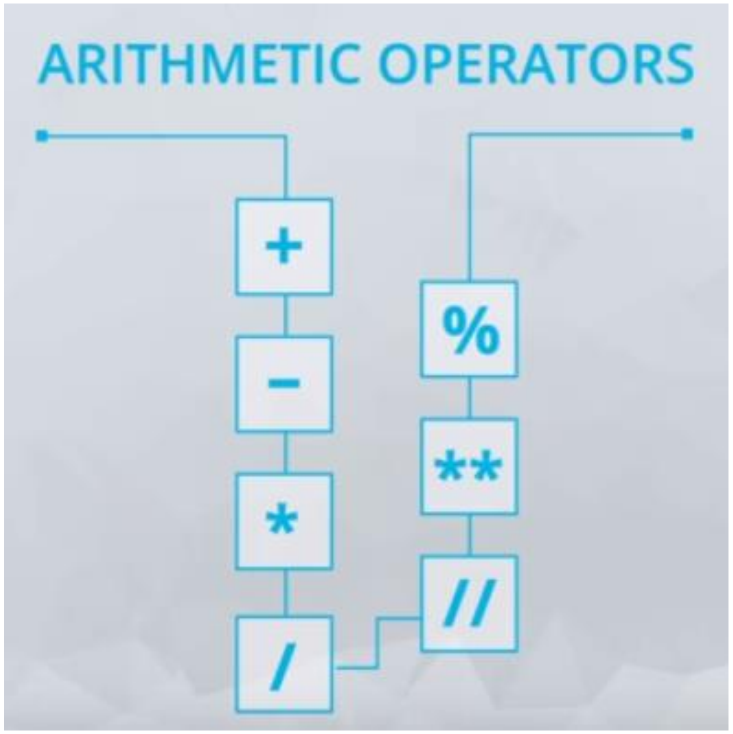
ASSIGNMENT OPERATORS

| SYMBOL | EXAMPLE | EQUIVALENT |
|---|---|---|
| = | x = 2 | x = 2 |
| += | x += 2 | x = x + 2 |
| −= | x −= 2 | x = x − 2 |

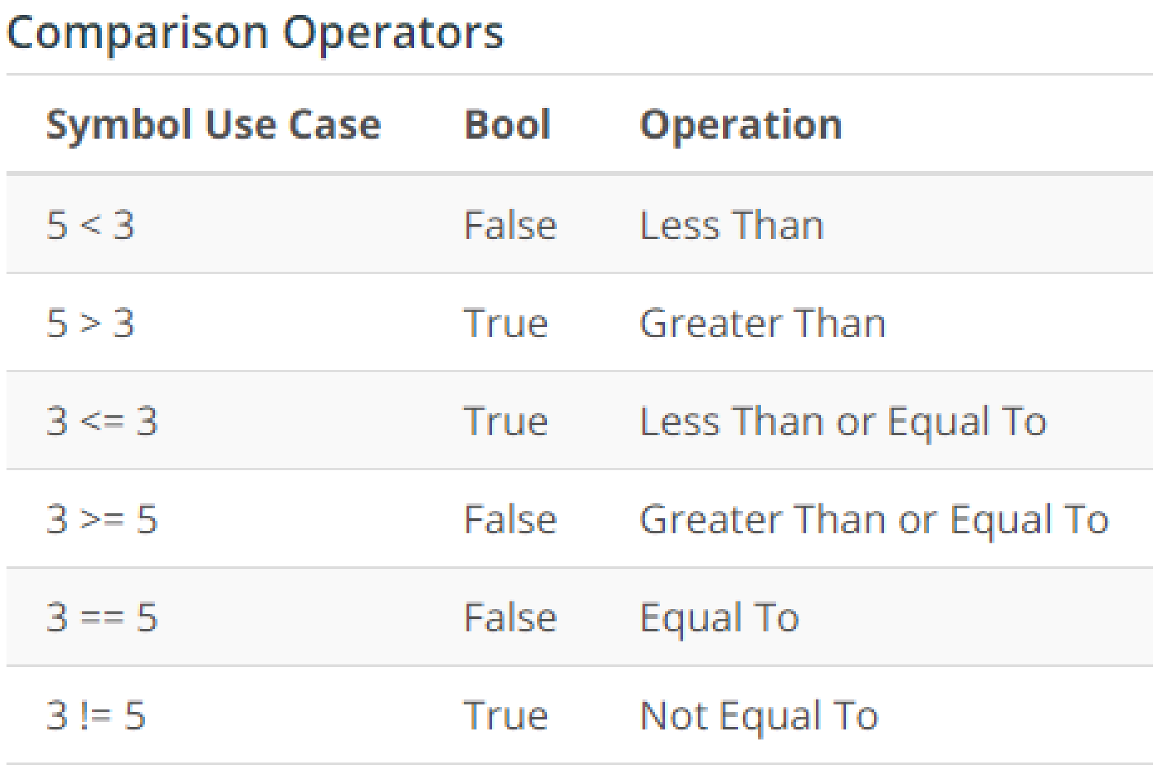
# Operators

1. **Arithmetic**
2. **Comparison**
3. **Logical**

## 1. Arithmetic Operators



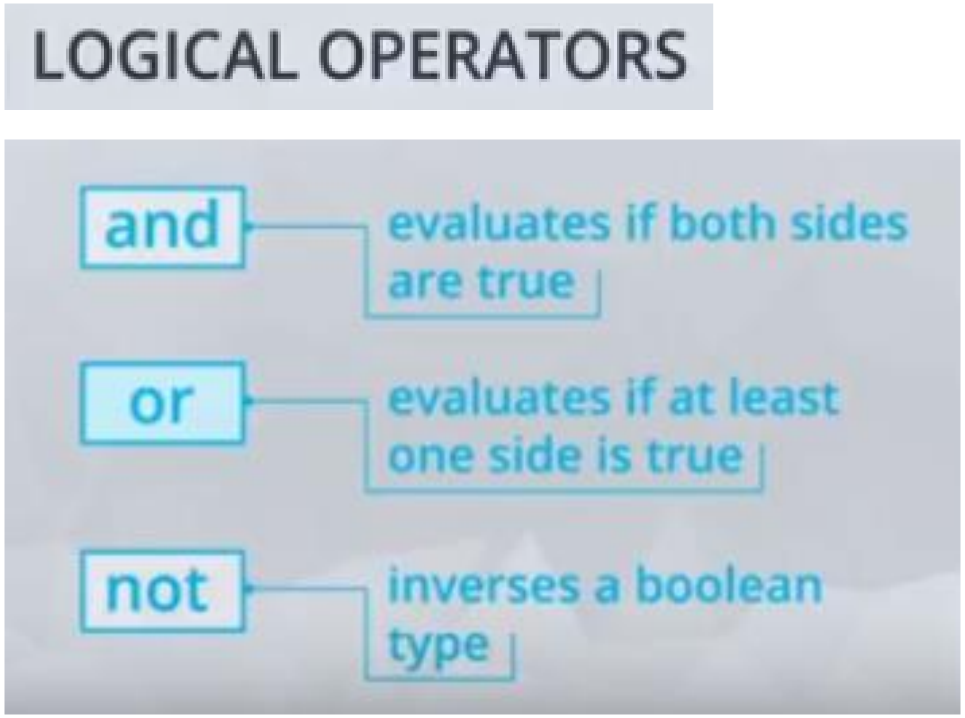
## 2. Comparison Operators



Comparison Operators

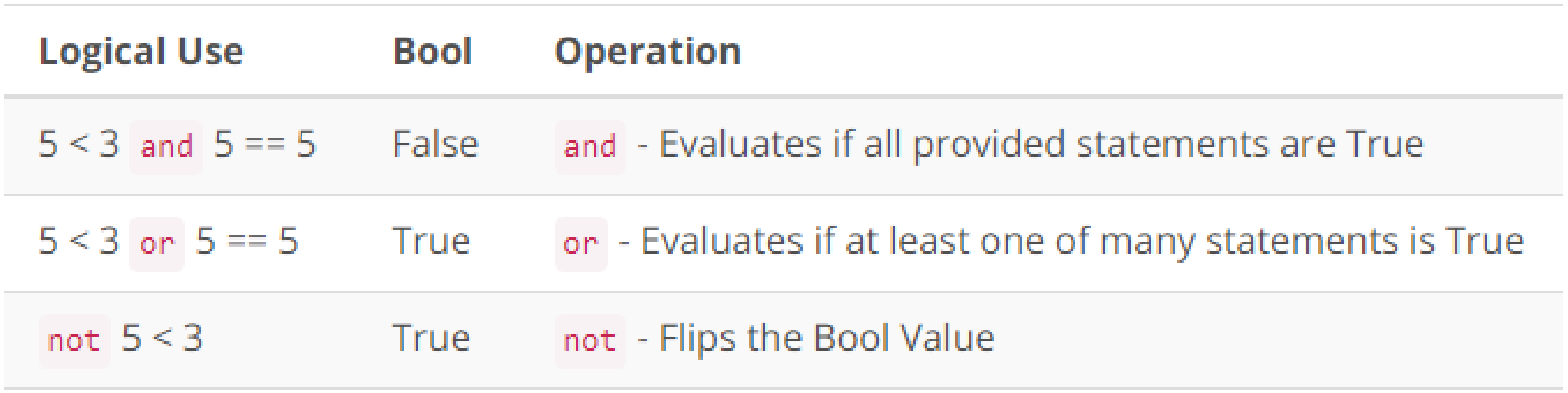
| Symbol Use Case | Bool | Operation |
|---|---|---|
| 5 < 3 | False | Less Than |
| 5 > 3 | True | Greater Than |
| 3 <= 3 | True | Less Than or Equal To |
| 3 >= 5 | False | Greater Than or Equal To |
| 3 == 5 | False | Equal To |
| 3 != 5 | True | Not Equal To |

## 3. Logical Operators


LOGICAL OPERATORS

| | |
|---|---|
| and | evaluates if both sides are true |
| or | evaluates if at least one side is true |
| not | inverses a boolean type |

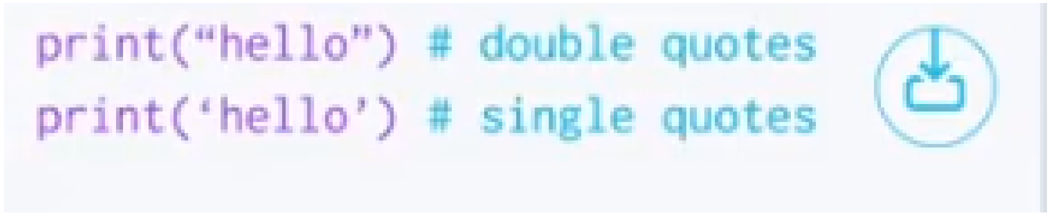
# Boolean Results – Comparison/Logical Operators



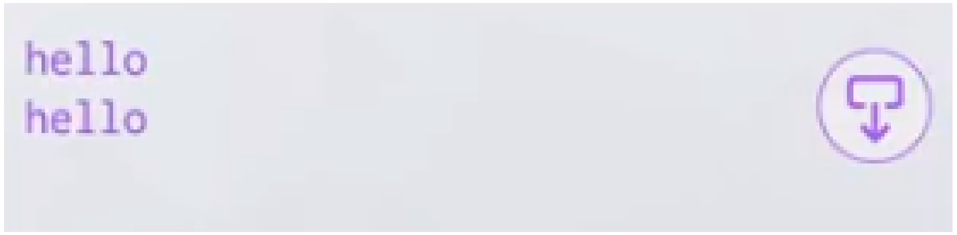
| Logical Use | Bool | Operation |
|---|---|---|
| 5 < 3 `and` 5 == 5 | False | `and` - Evaluates if all provided statements are True |
| 5 < 3 `or` 5 == 5 | True | `or` - Evaluates if at least one of many statements is True |
| `not` 5 < 3 | True | `not` - Flips the Bool Value |

## Strings

## String - Data Type

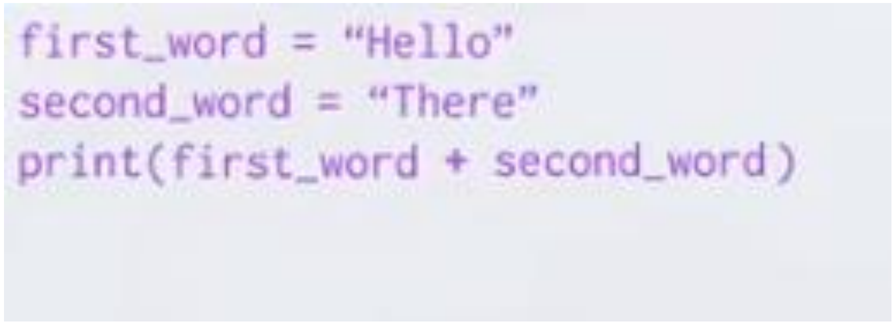


```
print("hello") # double quotes
print('hello') # single quotes
```

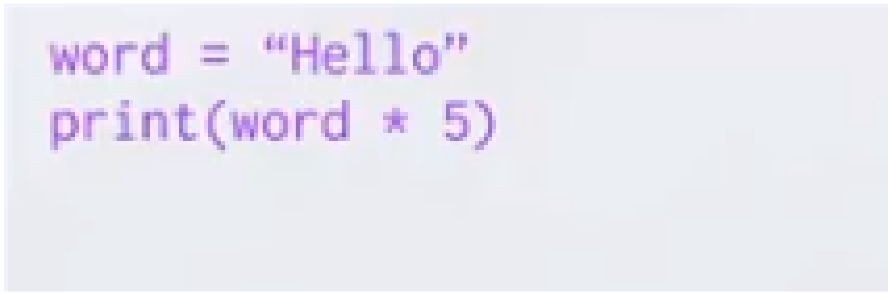


```
hello
hello
```

## String - Addition / Multiplication

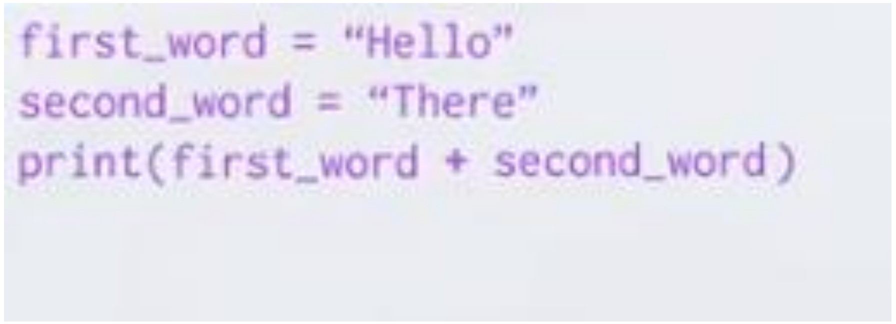


```
first_word = “Hello”
second_word = “There”
print(first_word + second_word)
```

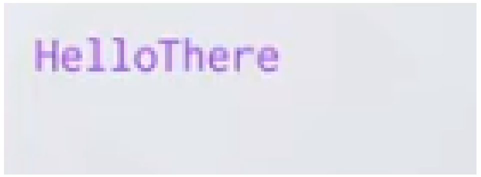


```
word = “Hello”
print(word * 5)
```

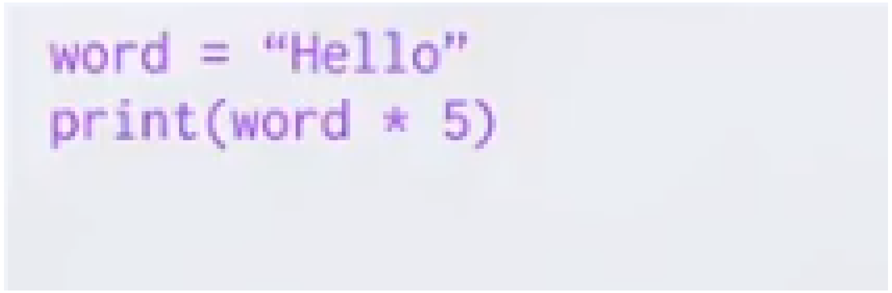
## String - Addition / Multiplication



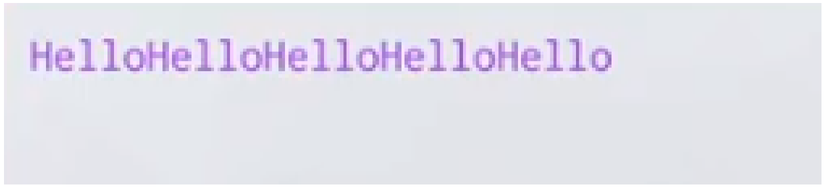
```
first_word = “Hello”
second_word = “There”
print(first_word + second_word)
```



```
HelloThere
```



```
word = “Hello”
print(word * 5)
```

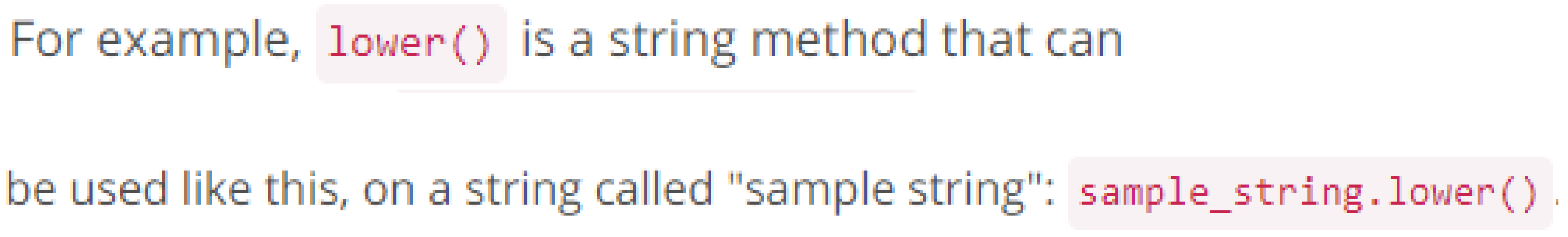


```
HelloHelloHelloHelloHello
```

## String Methods

**Methods** actually are functions that belong to an object/specific to the data type and are called using **dot notation**.

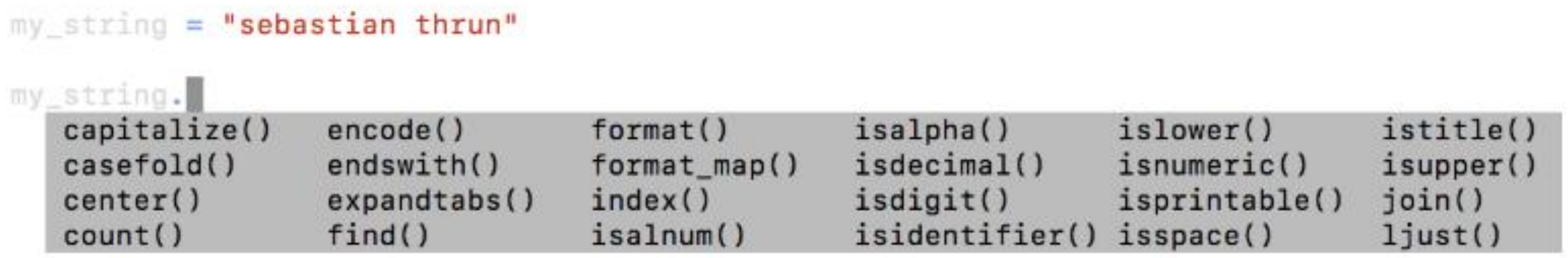
# String Methods



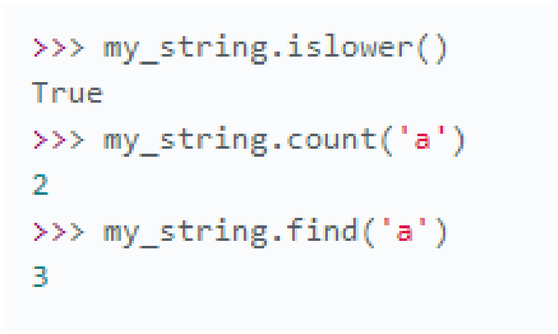
For example, `lower()` is a string method that can be used like this, on a string called "sample string": `sample_string.lower()`.

# String Methods

some methods that are possible with any string.



```
my_string = "sebastian thrun"

my_string.
  capitalize()  encode()      format()      isalpha()       islower()     istitle()
  casefold()    endswith()    format_map()  isdecimal()     isnumeric()   isupper()
  center()      expandtabs()  index()       isdigit()       isprintable() join()
  count()       find()        isalnum()     isidentifier()  isspace()     ljust()
```



```
>>> my_string.islower()
True
>>> my_string.count('a')
2
>>> my_string.find('a')
3
```

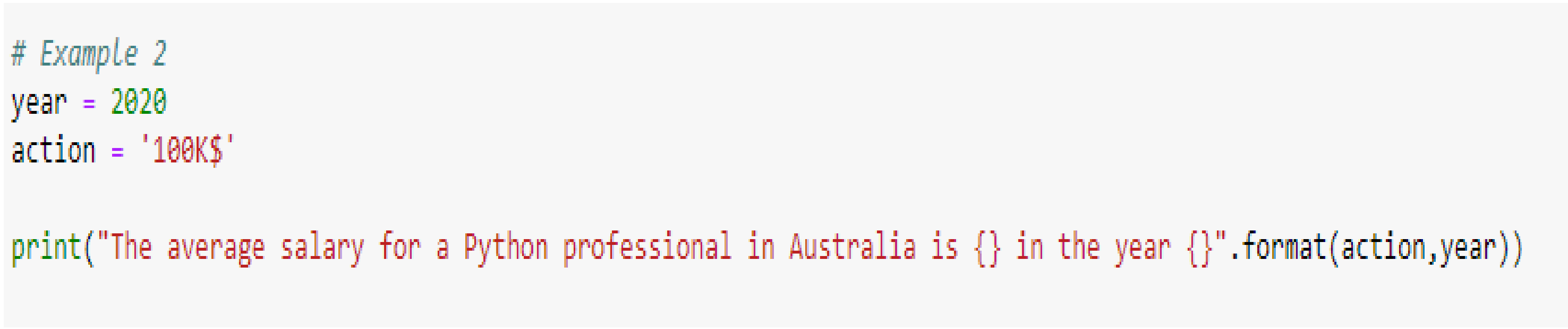
## String Methods

One important string method: `format()`

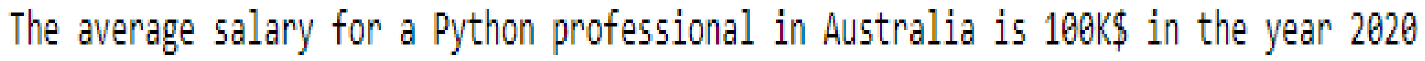
```
# Example 1
print("Python has a huge demand in the year {}".format('2020'))
```

```
Python has a huge demand in the year 2020
```

# String Methods

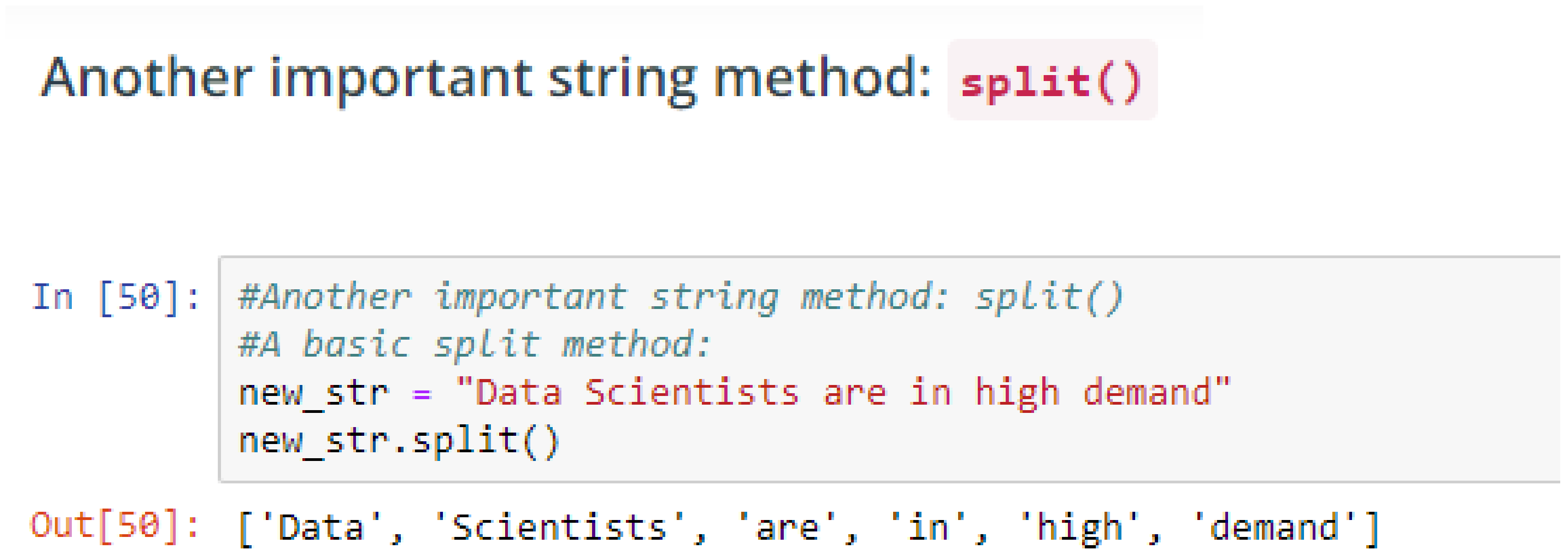


```python
# Example 2
year = 2020
action = '100K$'

print("The average salary for a Python professional in Australia is {} in the year {}".format(action,year))
```



```
The average salary for a Python professional in Australia is 100K$ in the year 2020
```

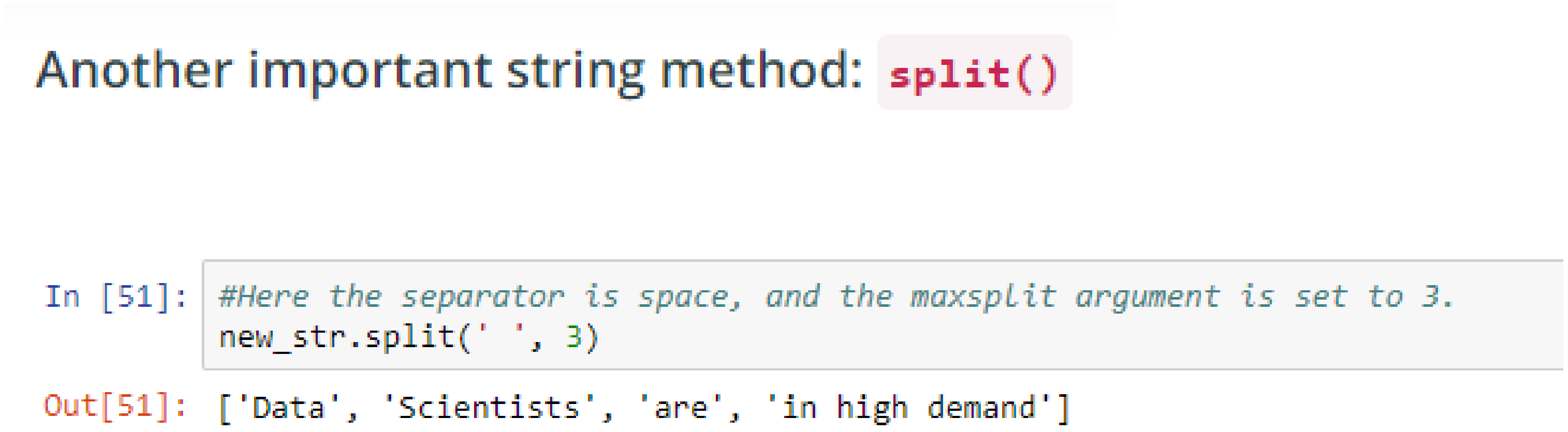
# String Methods



Another important string method: `split()`

```
In [50]: #Another important string method: split()
         #A basic split method:
         new_str = "Data Scientists are in high demand"
         new_str.split()

Out[50]: ['Data', 'Scientists', 'are', 'in', 'high', 'demand']
```

# String Methods



Another important string method: `split()`

```
In [51]: #Here the separator is space, and the maxsplit argument is set to 3.
         new_str.split(' ', 3)

Out[51]: ['Data', 'Scientists', 'are', 'in high demand']
```

## Type and Type Conversion

## Data structures

# Data structures

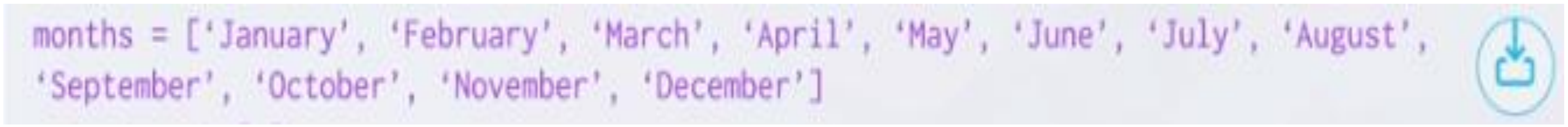
1. List
2. Tuples
3. Set
4. Dictionary

Also Compound Data structure

**Data structures** are containers that organize and group different data types together.

**List** is one of the most common and basic data structures in Python.

## List



```
months = [‘January’, ‘February’, ‘March’, ‘April’, ‘May’, ‘June’, ‘July’, ‘August’,
‘September’, ‘October’, ‘November’, ‘December’]
```

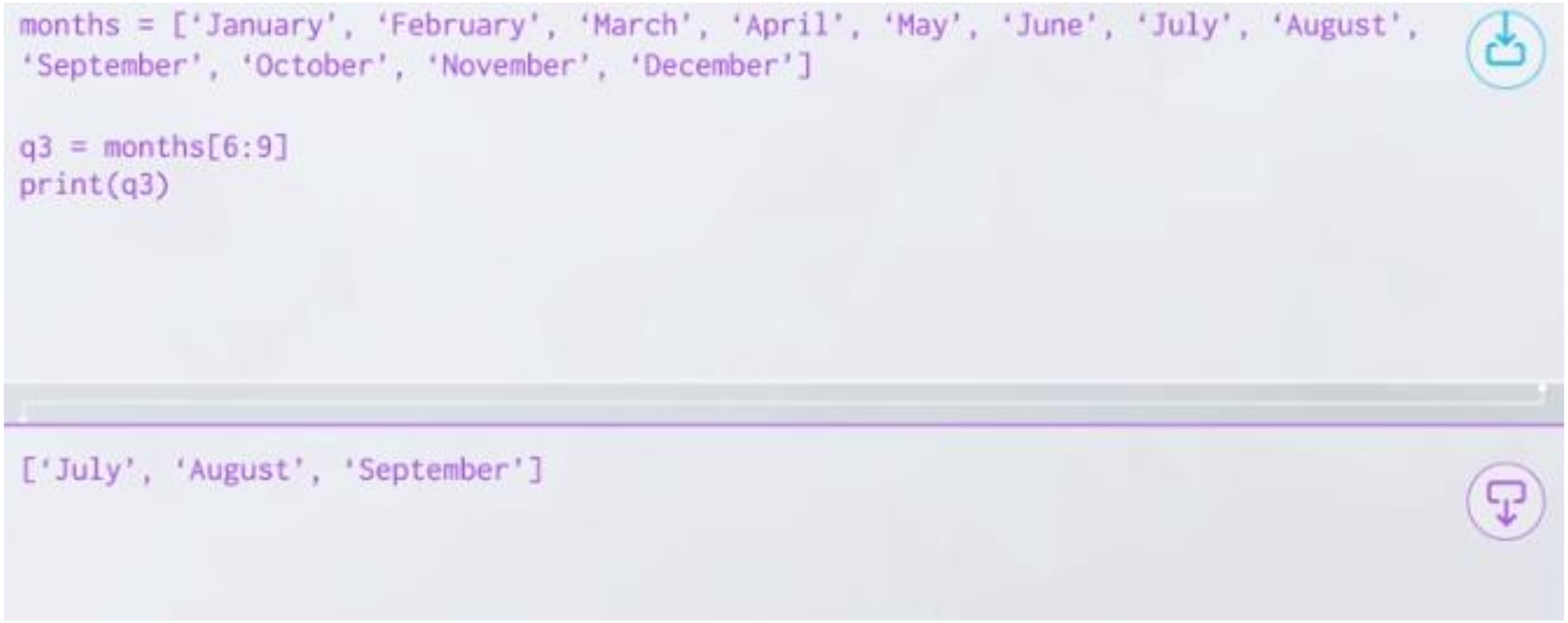
## List - Indexing

```
months = [‘January’, ‘February’, ‘March’, ‘April’, ‘May’, ‘June’, ‘July’, ‘August’,
‘September’, ‘October’, ‘November’, ‘December’]
```

```
print(months[0])
print(months[1])
print(months[7])
```

```
January
February
August
```

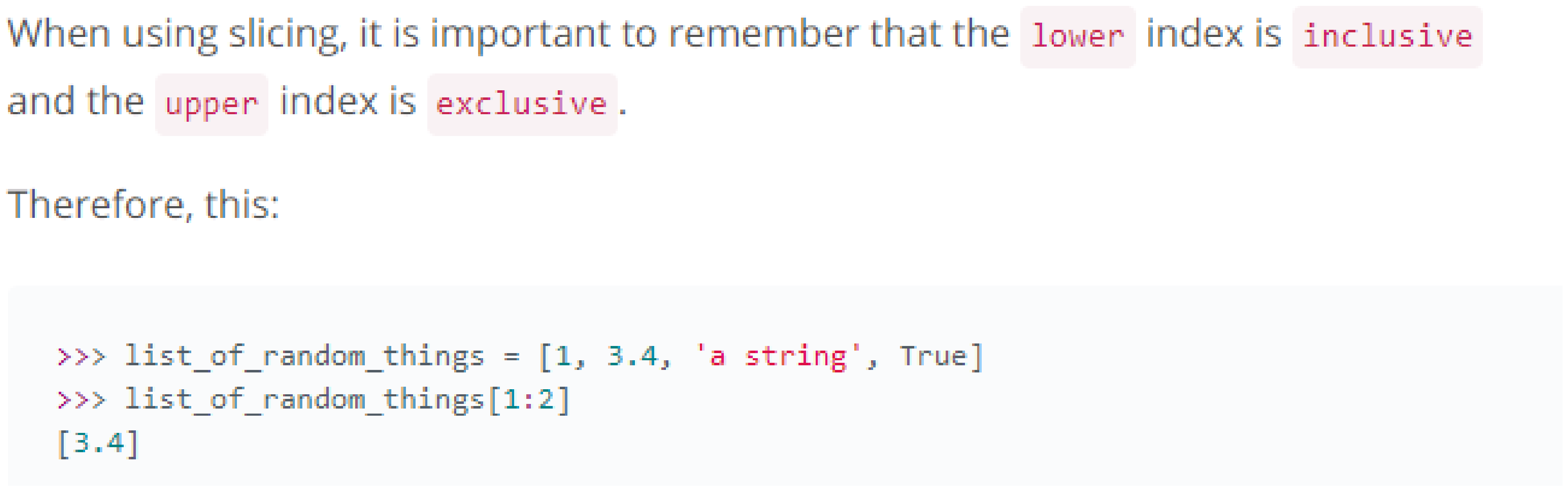
# Slice and Dice with Lists



```
months = ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
'September', 'October', 'November', 'December']

q3 = months[6:9]
print(q3)
```

```
['July', 'August', 'September']
```

# Slice and Dice with Lists

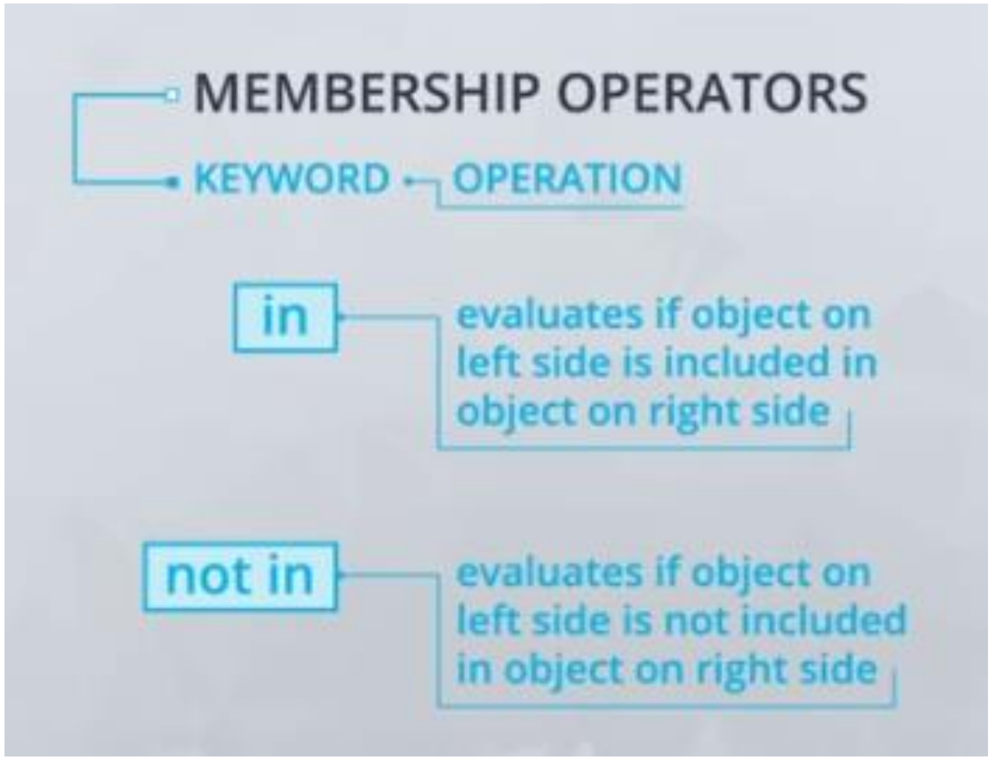


When using slicing, it is important to remember that the `lower` index is `inclusive` and the `upper` index is `exclusive`.

Therefore, this:

```
>>> list_of_random_things = [1, 3.4, 'a string', True]
>>> list_of_random_things[1:2]
[3.4]
```

# Membership Operators

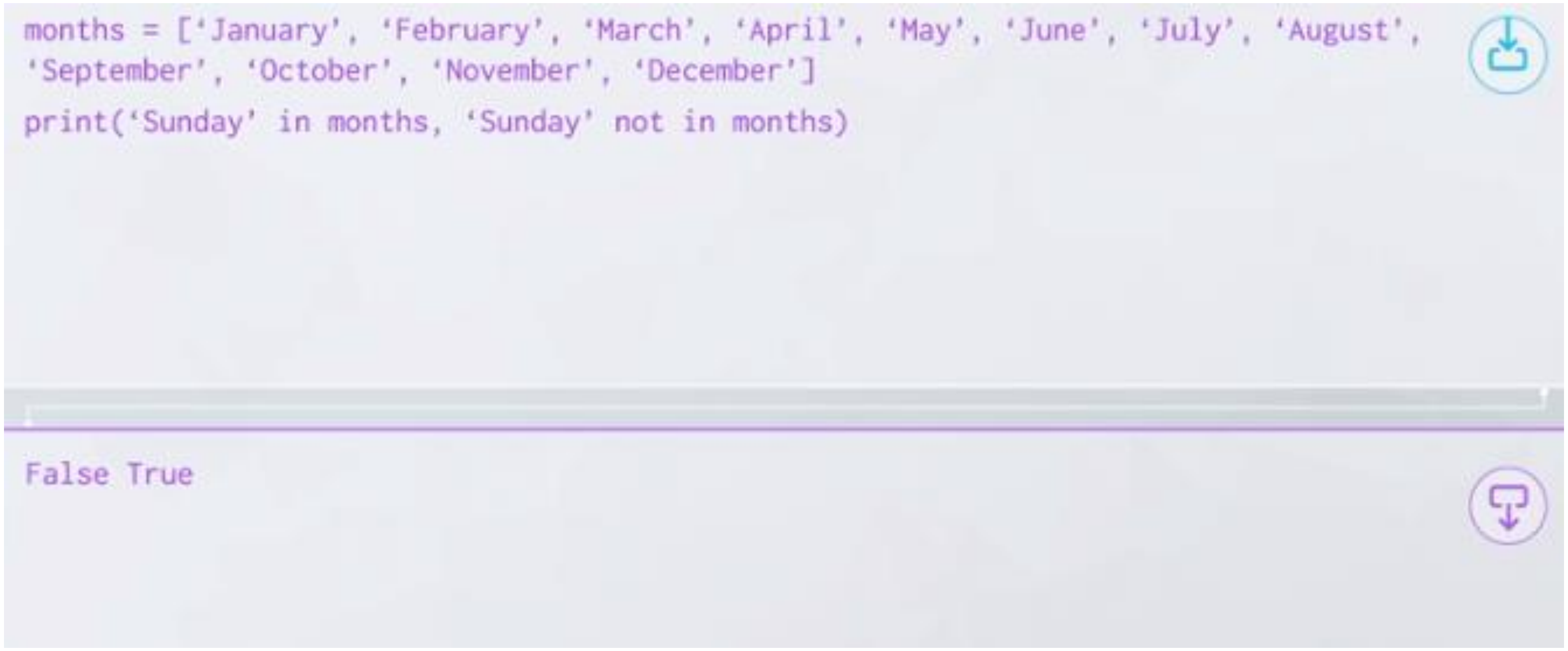

MEMBERSHIP OPERATORS

KEYWORD ← OPERATION

in — evaluates if object on left side is included in object on right side

not in — evaluates if object on left side is not included in object on right side

# List and Membership Operators



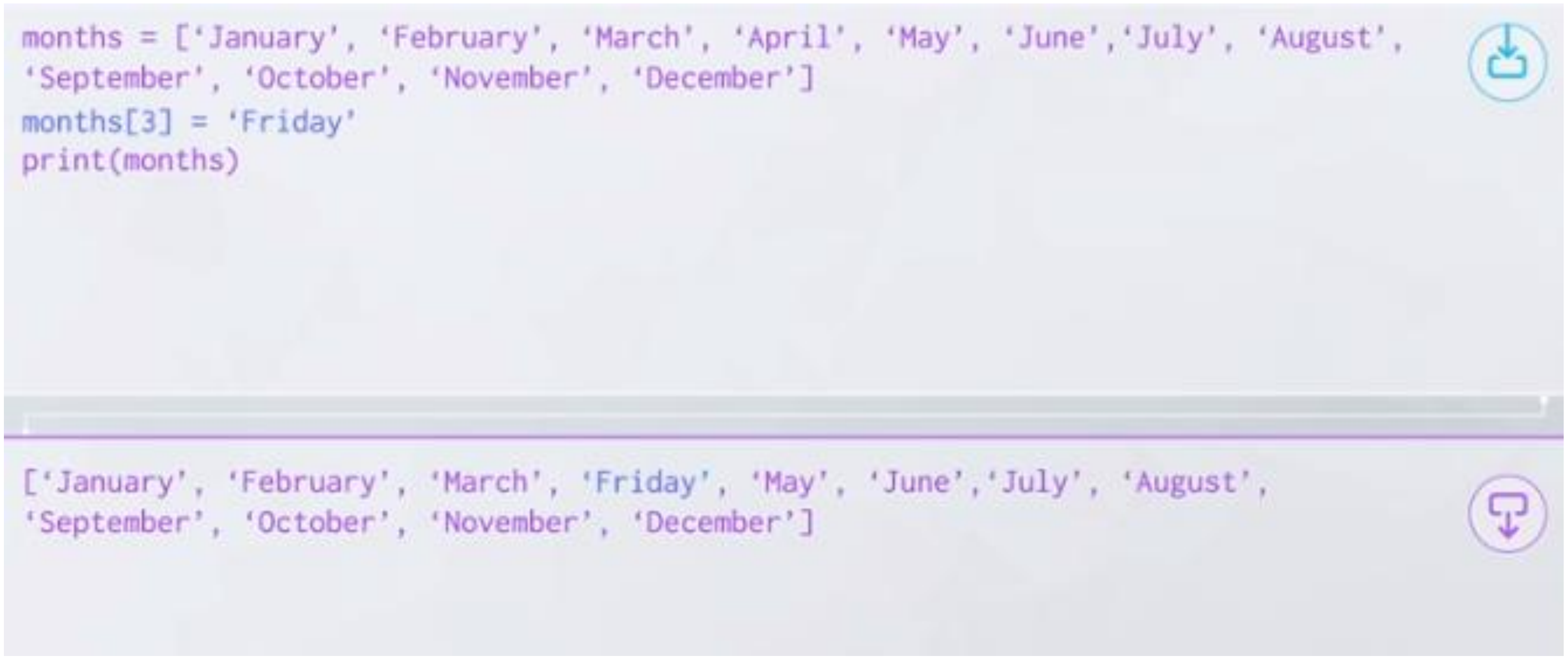
```
months = ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
'September', 'October', 'November', 'December']
print('Sunday' in months, 'Sunday' not in months)
```

```
False True
```

## Mutability

- **Mutability** is about whether or not we can change an object once it has been created.
- If an object can be changed, then it is called **mutable**.
- If an object cannot be changed without creating a completely new object then the object is considered **immutable**.

# Mutability

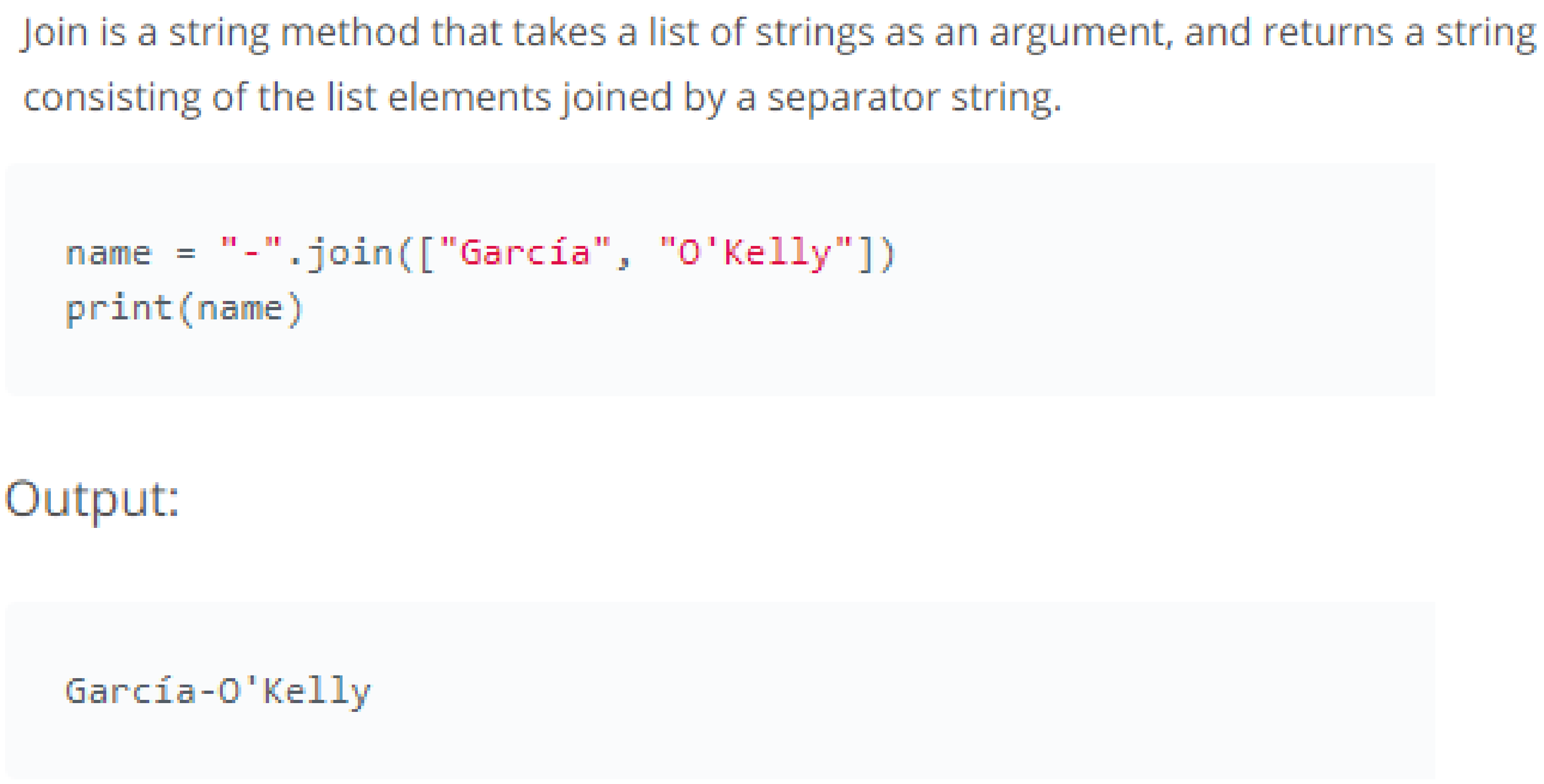


```
months = ['January', 'February', 'March', 'April', 'May', 'June','July', 'August',
'September', 'October', 'November', 'December']
months[3] = 'Friday'
print(months)
```

```
['January', 'February', 'March', 'Friday', 'May', 'June','July', 'August',
'September', 'October', 'November', 'December']
```

# List Methods

**Join method**



Join is a string method that takes a list of strings as an argument, and returns a string consisting of the list elements joined by a separator string.

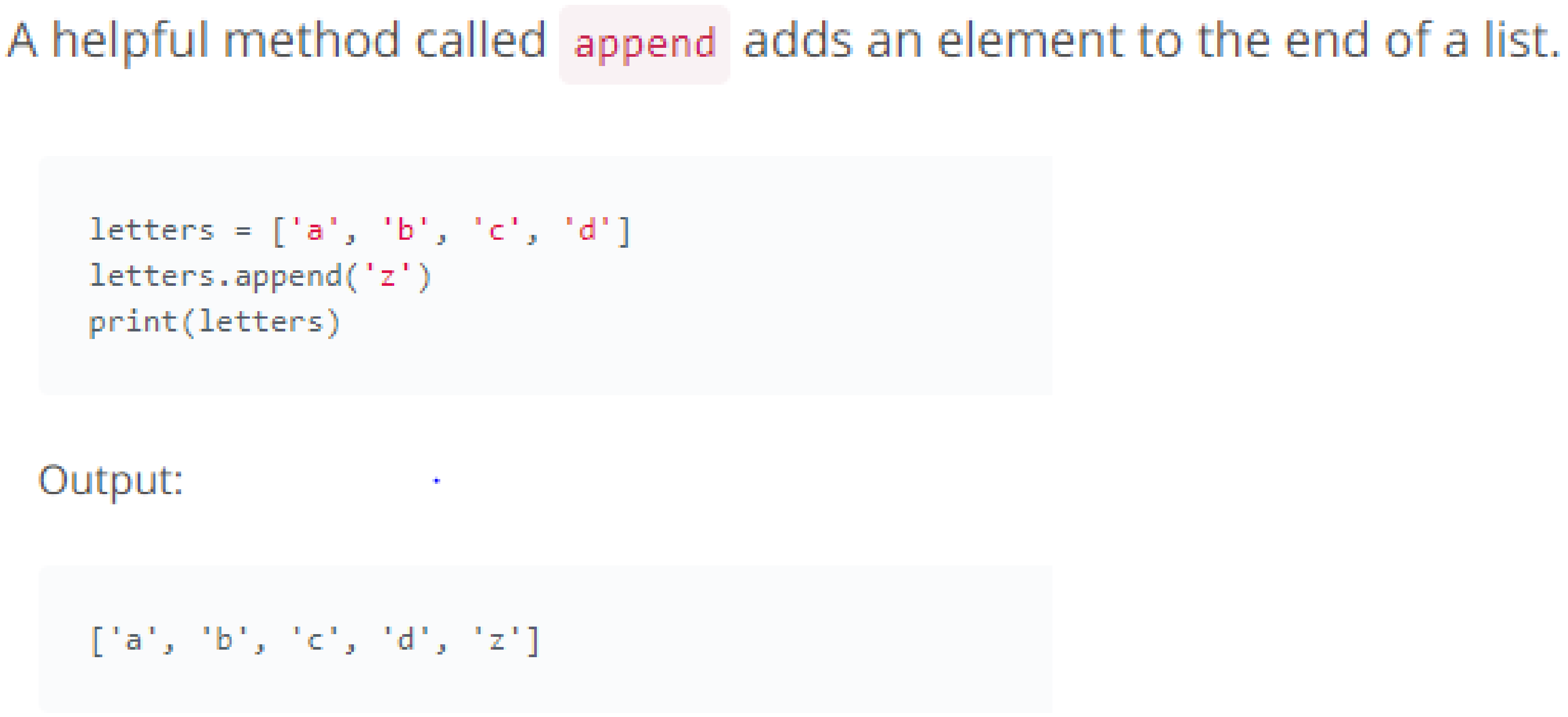
```
name = "-".join(["García", "O'Kelly"])
print(name)
```

Output:

```
García-O'Kelly
```

## List Methods

**append method**



A helpful method called `append` adds an element to the end of a list.

```
letters = ['a', 'b', 'c', 'd']
letters.append('z')
print(letters)
```

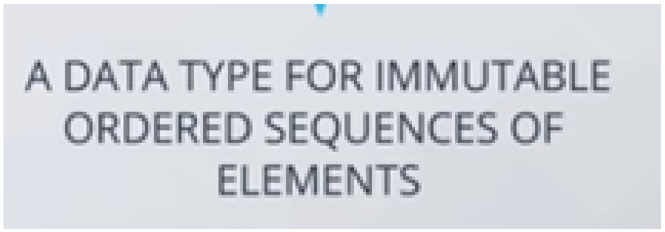
Output:

```
['a', 'b', 'c', 'd', 'z']
```

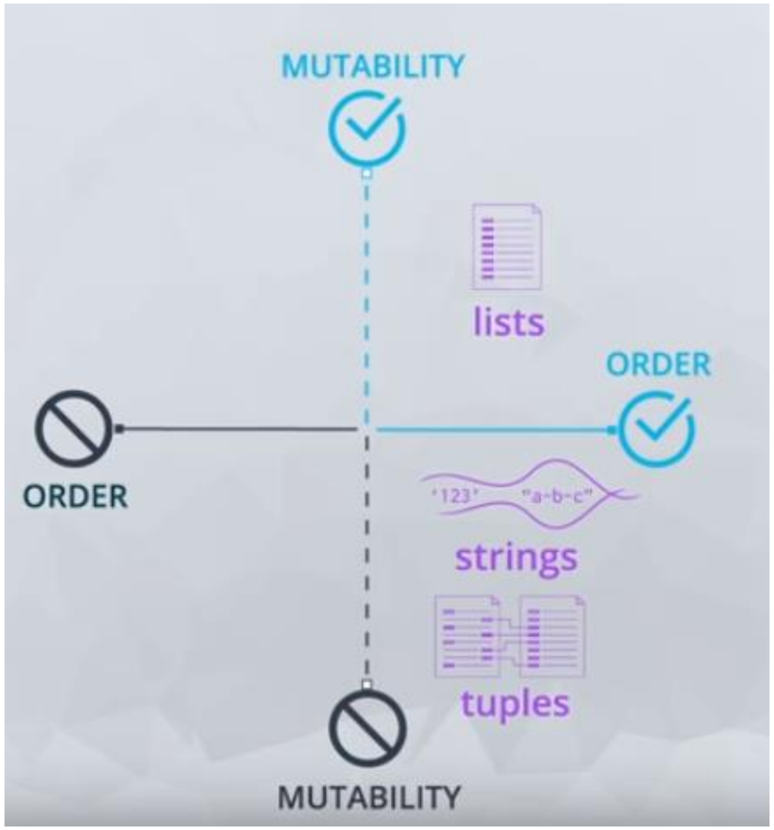
## Tuples

Tuples are immutable ordered sequence of elements

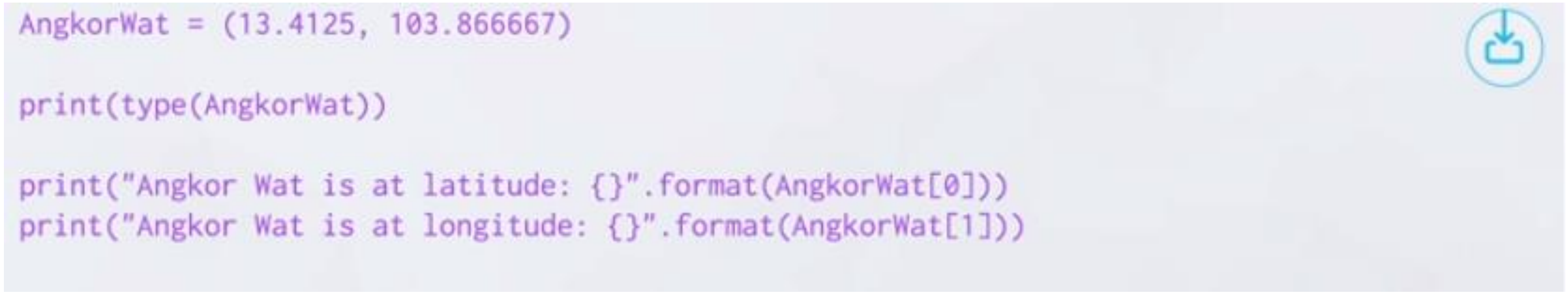
# Tuples



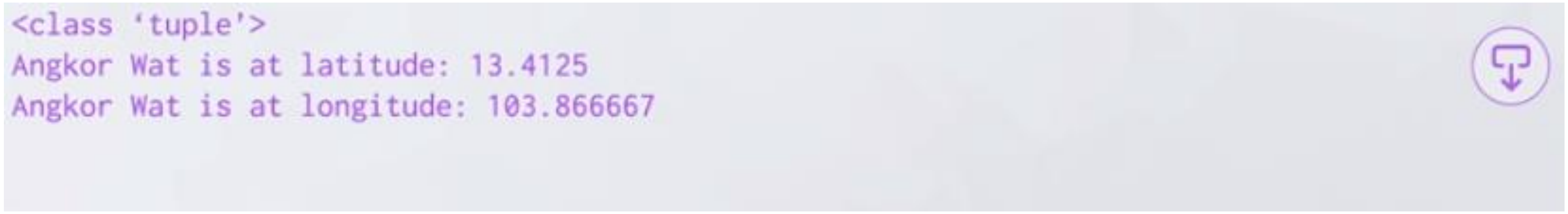
A DATA TYPE FOR IMMUTABLE ORDERED SEQUENCES OF ELEMENTS



# Tuples - Indexing



```
AngkorWat = (13.4125, 103.866667)

print(type(AngkorWat))

print("Angkor Wat is at latitude: {}".format(AngkorWat[0]))
print("Angkor Wat is at longitude: {}".format(AngkorWat[1]))
```

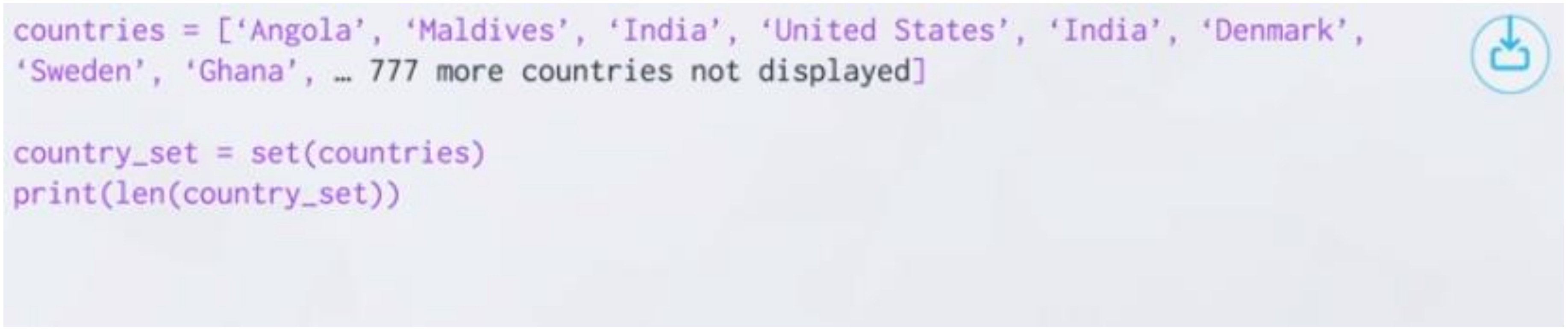


```
<class 'tuple'>
Angkor Wat is at latitude: 13.4125
Angkor Wat is at longitude: 103.866667
```

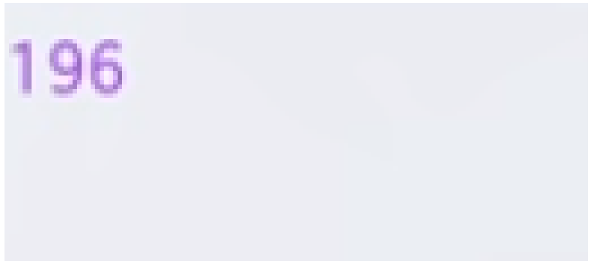
## SET

Sets are mutable ordered sequence of elements

## SET

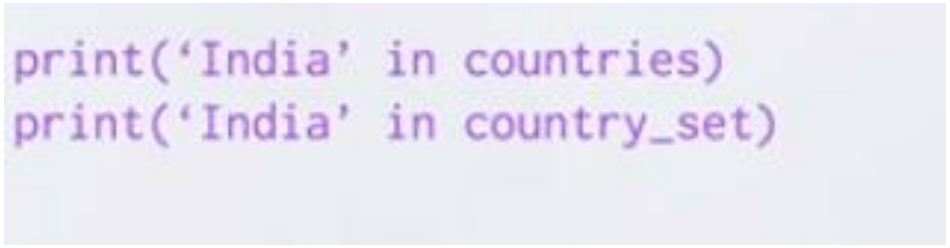


```
countries = [‘Angola’, ‘Maldives’, ‘India’, ‘United States’, ‘India’, ‘Denmark’,
‘Sweden’, ‘Ghana’, … 777 more countries not displayed]

country_set = set(countries)
print(len(country_set))
```

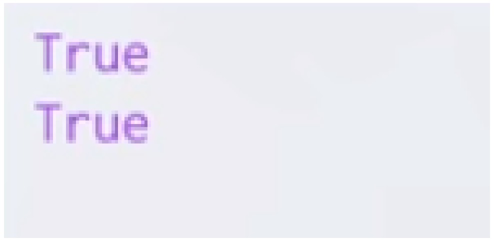


```
196
```

## SET and Membership operators



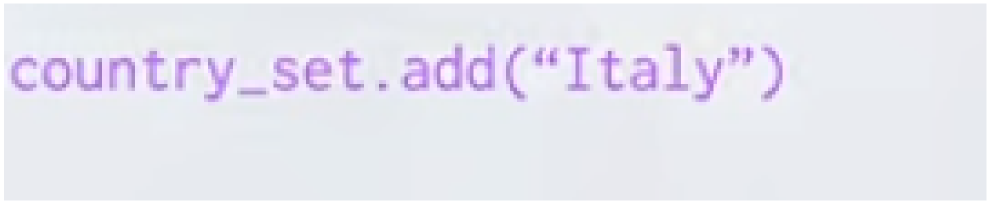
```
print('India' in countries)
print('India' in country_set)
```



```
True
True
```

## SET Methods

**add method**

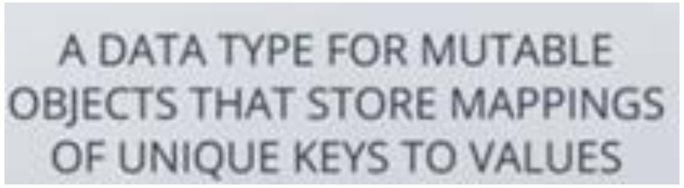


```
country_set.add("Italy")
```

## SET Methods

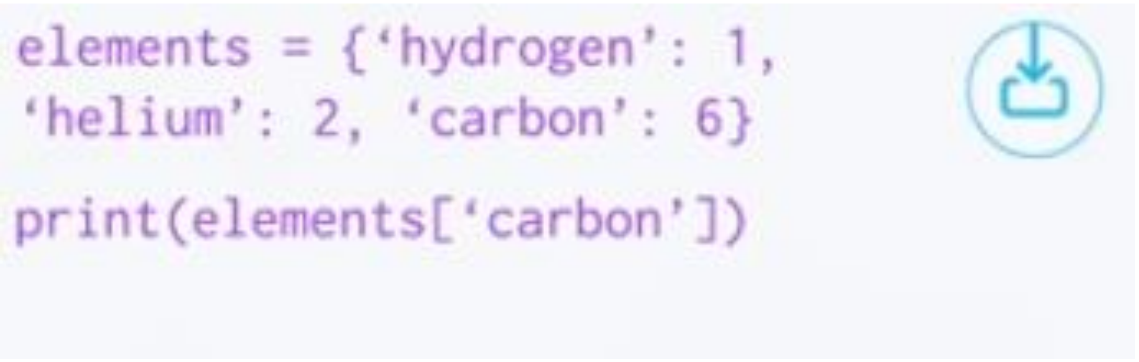
Methods like union, intersection, and difference are easy to perform with sets, and are much faster than such operators with other containers

# Dictionaries

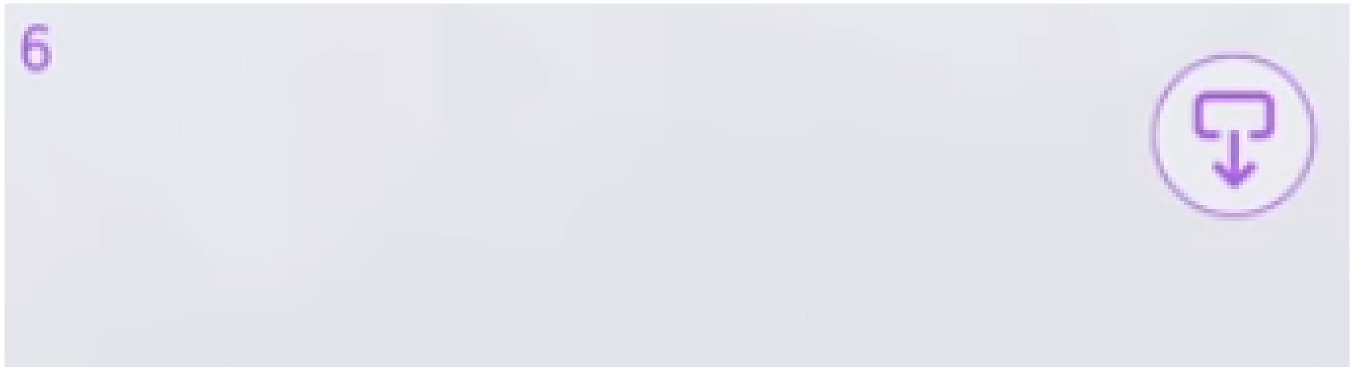

A DATA TYPE FOR MUTABLE OBJECTS THAT STORE MAPPINGS OF UNIQUE KEYS TO VALUES

A **dictionary** is a mutable data type that stores mappings of unique keys to values

## Dictionaries Indexing(Key)

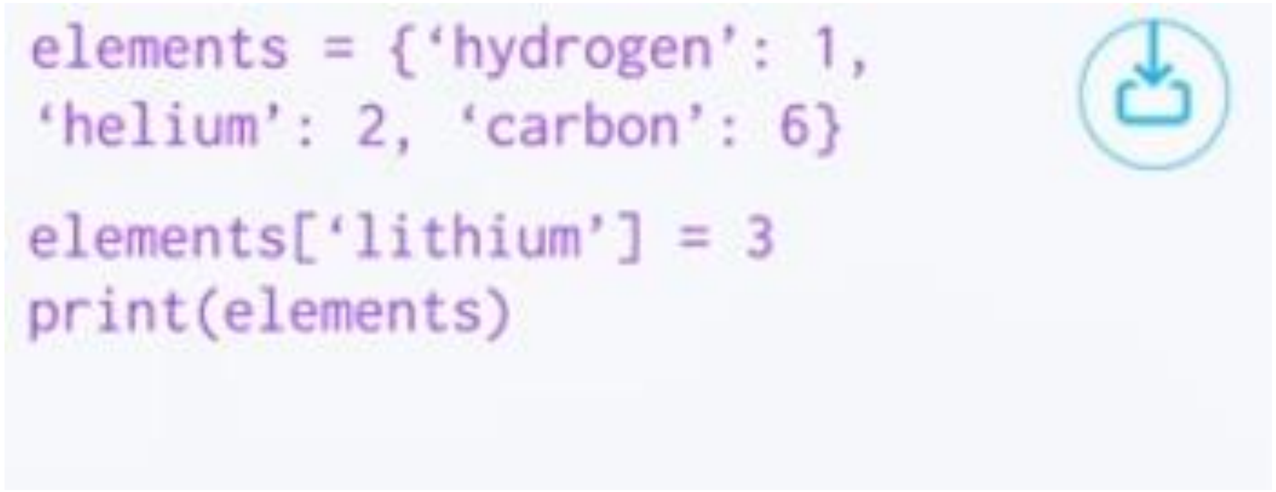


```
elements = {'hydrogen': 1,
'helium': 2, 'carbon': 6}

print(elements['carbon'])
```

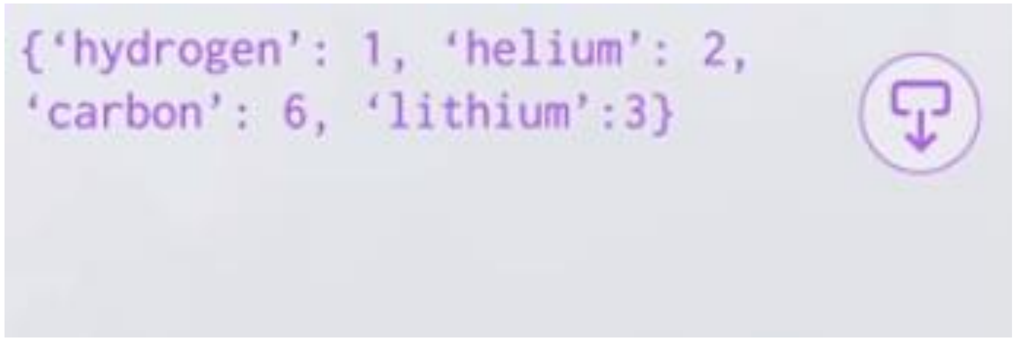


```
6
```

## Dictionaries - Adding Key/Values

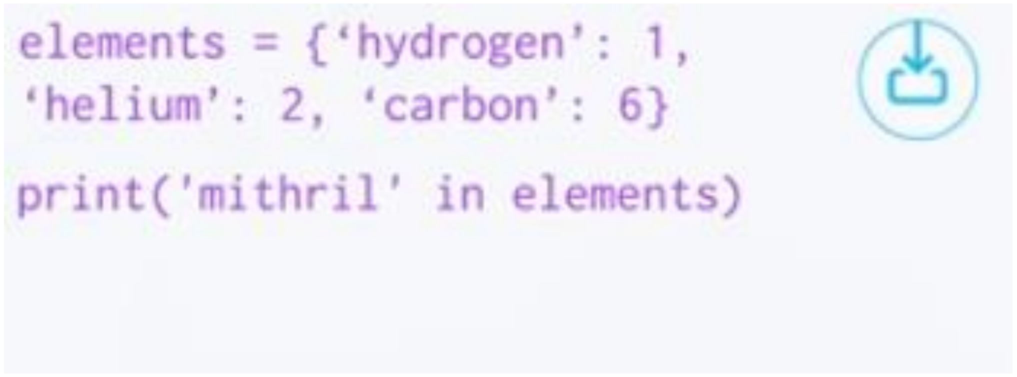


```
elements = {'hydrogen': 1,
'helium': 2, 'carbon': 6}

elements['lithium'] = 3
print(elements)
```

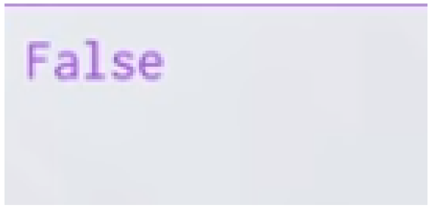


```
{'hydrogen': 1, 'helium': 2,
'carbon': 6, 'lithium':3}
```

# Dictionaries and Membership Operators



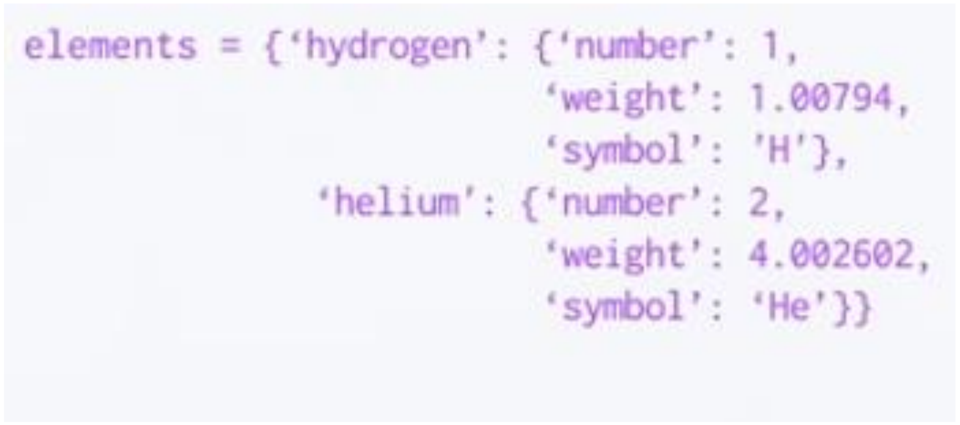
```
elements = {‘hydrogen’: 1,
‘helium’: 2, ‘carbon’: 6}
print('mithril' in elements)
```



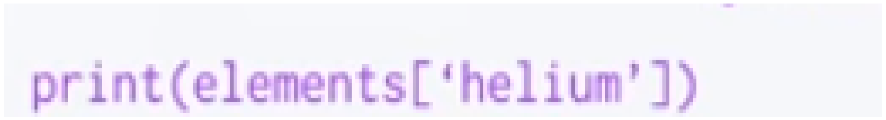
```
False
```

# Compound Data Structures

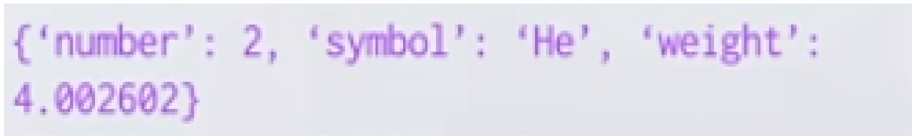
We can include containers in other containers to create compound data structures.



```
elements = {'hydrogen': {'number': 1,
                         'weight': 1.00794,
                         'symbol': 'H'},
            'helium': {'number': 2,
                       'weight': 4.002602,
                       'symbol': 'He'}}
```



```
print(elements['helium'])
```



```
{'number': 2, 'symbol': 'He', 'weight':
4.002602}
```

# Compound Data Structures

```
elements = {‘hydrogen’: {‘number’: 1,
                         ‘weight’: 1.00794,
                         ‘symbol’: ’H’},
            ‘helium’: {‘number’: 2,
                       ‘weight’: 4.002602,
                       ‘symbol’: ‘He’}}
```

```
print(elements[‘helium’][‘weight’])
```

```
4.002602
```

| Data Structure | Ordered | Mutable | Constructor | Example |
|---|---|---|---|---|
| list | Yes | Yes | `[ ]` or `list()` | [5, 'yes', 5.7] |
| tuple | Yes | No | `( )` or `tuple()` | (5, 'yes', 5.7) |
| set | No | Yes | `{ }` or `set()` | {5, 'yes', 5.7} |
| dictionary | No | Keys: No | `{ }` or `dict()` | {'Jun':75, 'Jul':89} |

## Control Flow

- Conditional Statements
- Loops - For / While
- Break and Continue

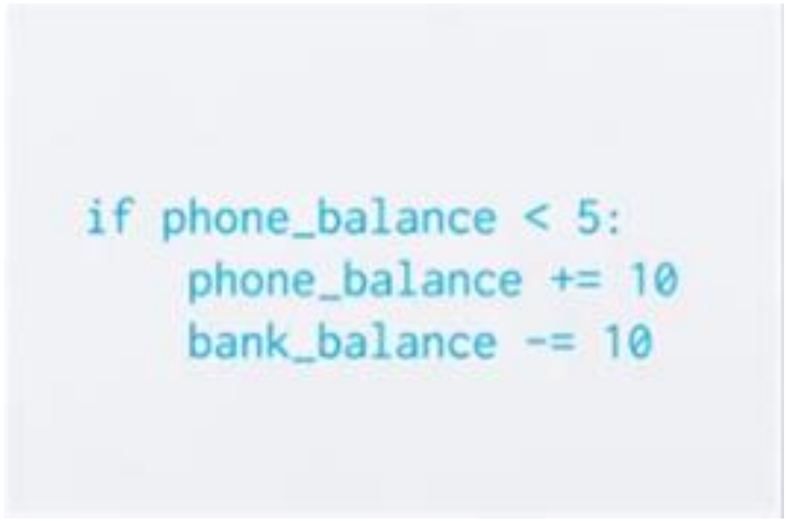
## Conditional Statement

## - If Statement

## If Statement

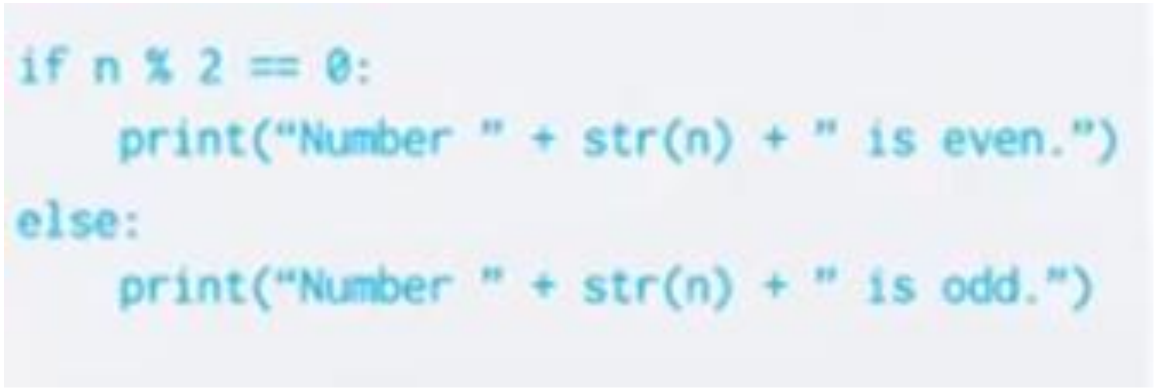
An if statement is a conditional statement that runs or skips code based on whether a condition is true or false.

## If Statement



```
if phone_balance < 5:
    phone_balance += 10
    bank_balance -= 10
```

## If - else Statement



```
if n % 2 == 0:
    print("Number " + str(n) + " is even.")
else:
    print("Number " + str(n) + " is odd.")
```

## If - elif - else Statement

## Loops

- For
- While

## For Loop

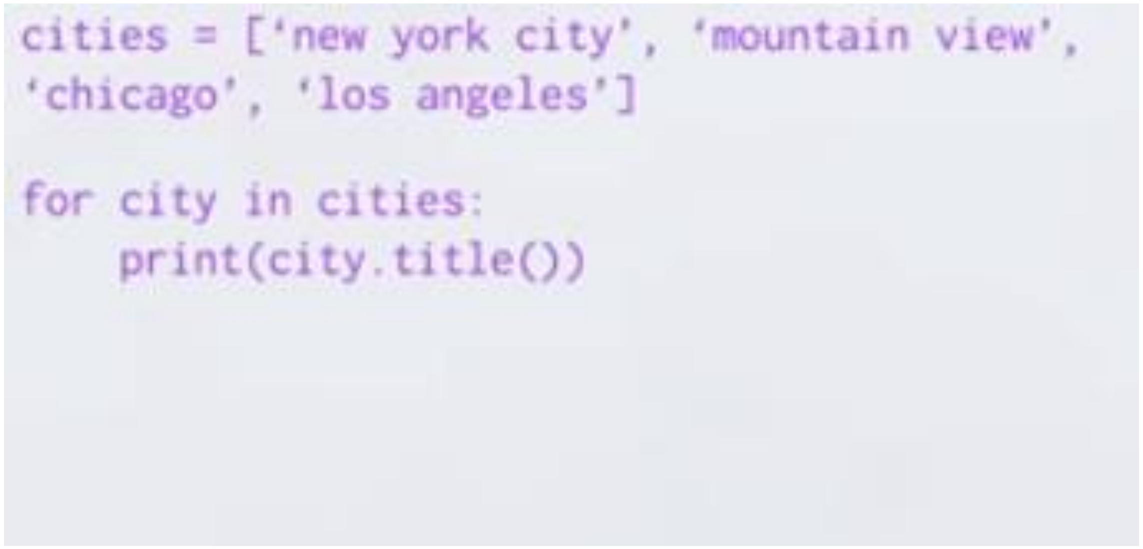
For loop is used to iterate or do something repeatedly, over an **iterable**.

## For Loop

ITERABLE

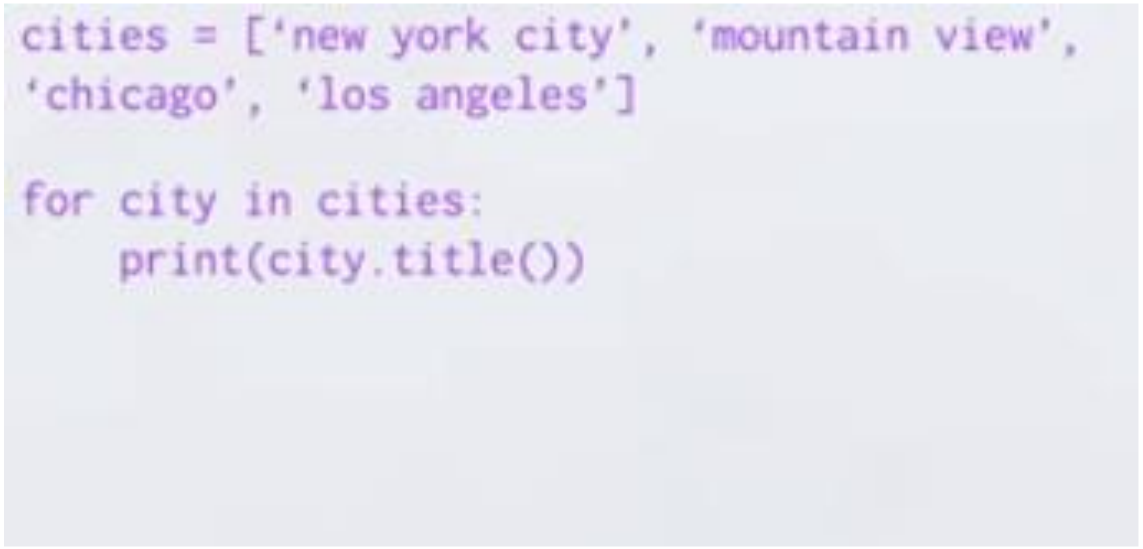
AN OBJECT THAT CAN RETURN
ONE OF ITS ELEMENTS AT A TIME

## For Loop

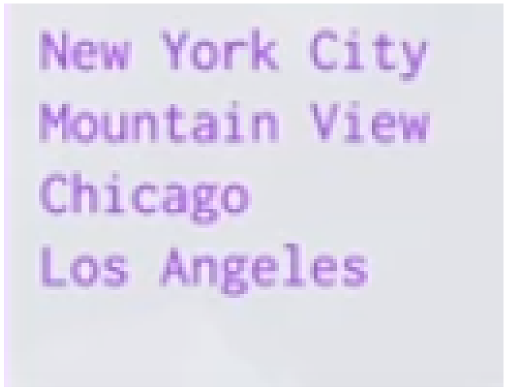


```
cities = [‘new york city’, ‘mountain view’,
‘chicago’, ‘los angeles’]

for city in cities:
    print(city.title())
```

# For Loop



```
cities = ['new york city', 'mountain view',
'chicago', 'los angeles']

for city in cities:
    print(city.title())
```



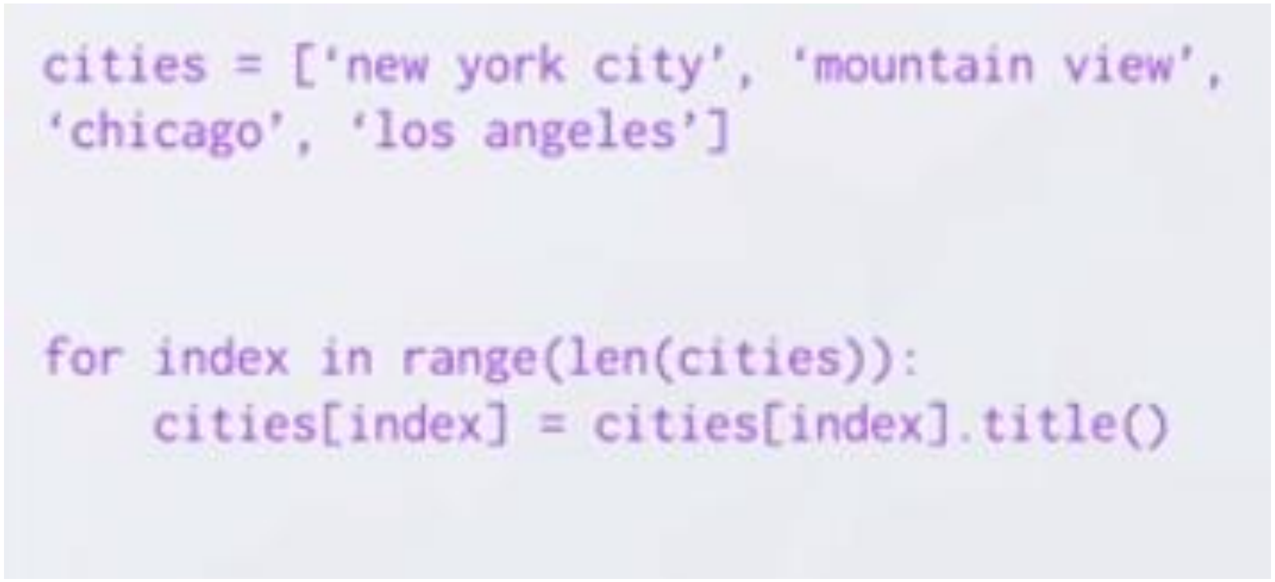
```
New York City
Mountain View
Chicago
Los Angeles
```

# For Loop

**How will you modify a list in a for loop without creating a new list ?**

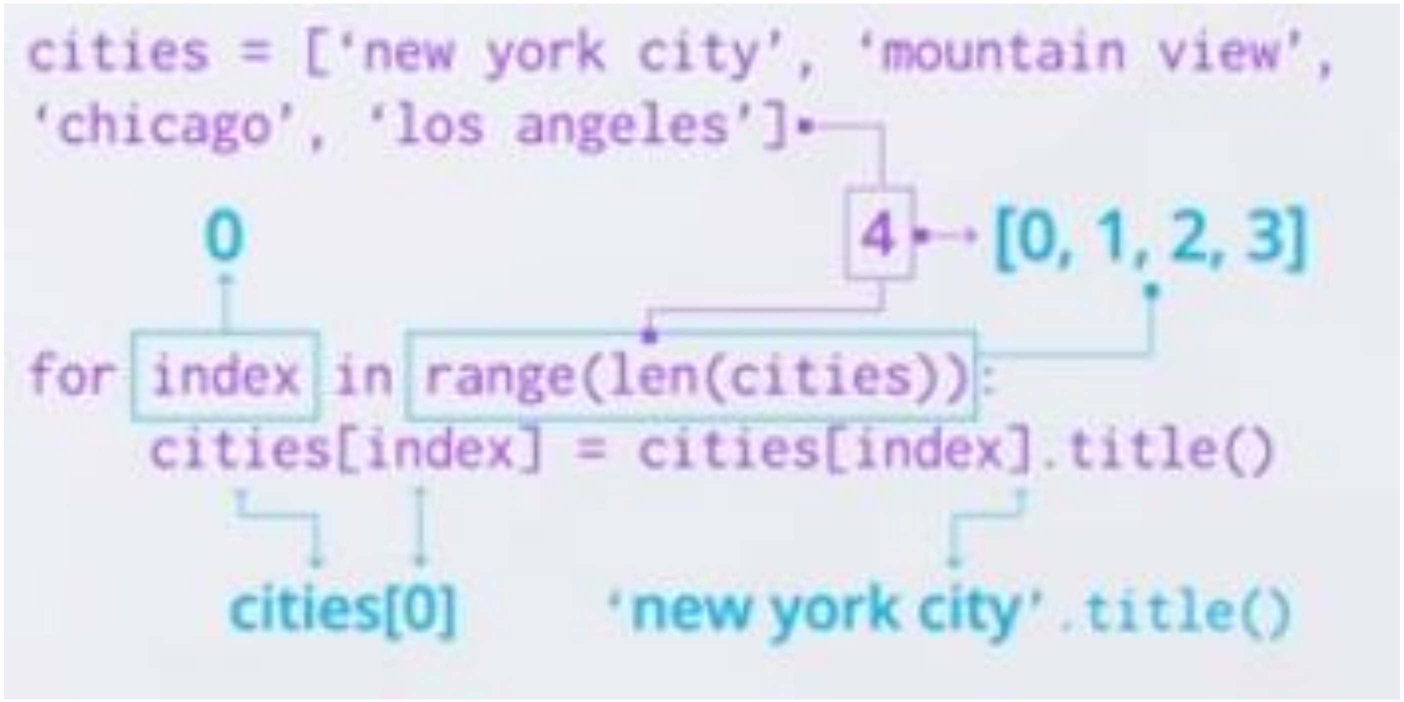
**HINT: Use the range function - Check out**

## For Loops to modify the same list



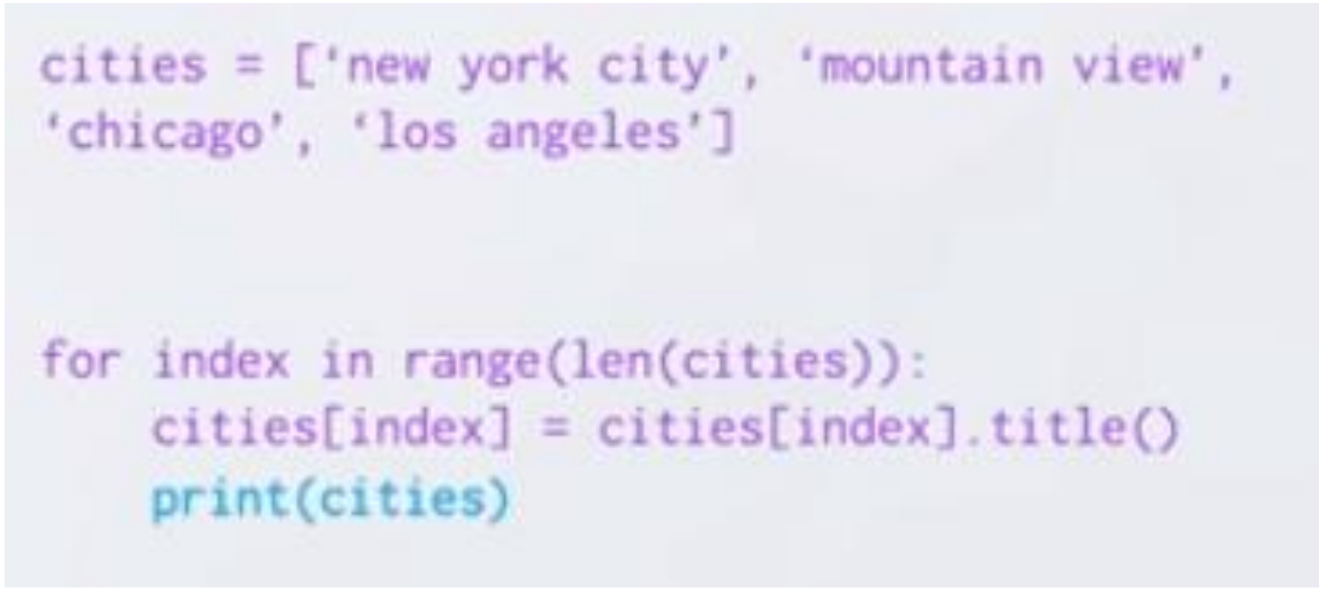
```
cities = ['new york city', 'mountain view',
'chicago', 'los angeles']


for index in range(len(cities)):
    cities[index] = cities[index].title()
```

# For Loops to modify a list

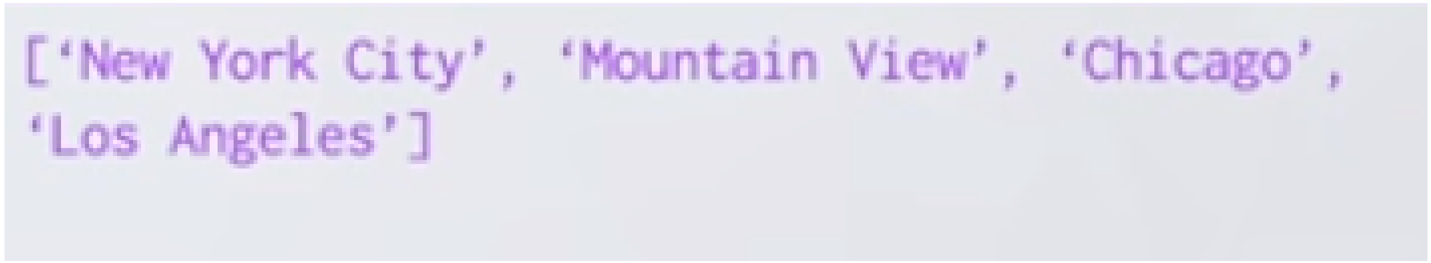


```
cities = ['new york city', 'mountain view',
'chicago', 'los angeles']

for index in range(len(cities)):
    cities[index] = cities[index].title()
```

# For Loops to modify a list



```
cities = ['new york city', 'mountain view',
'chicago', 'los angeles']


for index in range(len(cities)):
    cities[index] = cities[index].title()
    print(cities)
```



```
['New York City', 'Mountain View', 'Chicago',
'Los Angeles']
```

## For Loops - Iterating Dictionaries

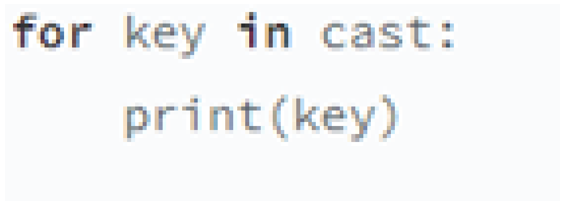
## For Loops - Iterating Dictionaries

```
cast = {
           "Jerry Seinfeld": "Jerry Seinfeld",
           "Julia Louis-Dreyfus": "Elaine Benes",
           "Jason Alexander": "George Costanza",
           "Michael Richards": "Cosmo Kramer"
       }
```

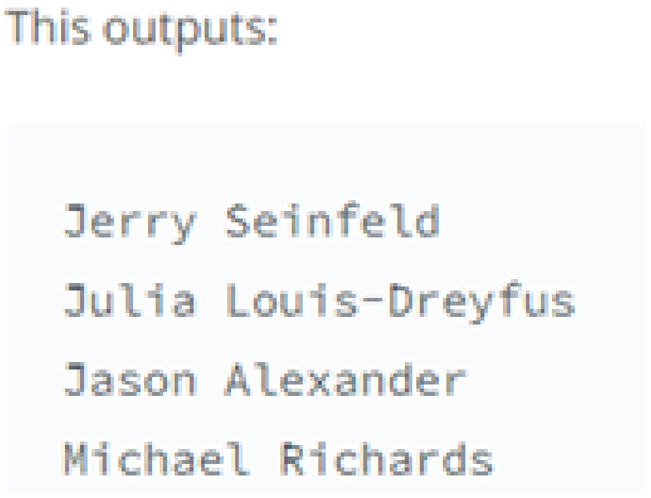
### How to get the below output:

```
Actor: Jerry Seinfeld     Role: Jerry Seinfeld
Actor: Julia Louis-Dreyfus     Role: Elaine Benes
Actor: Jason Alexander     Role: George Costanza
Actor: Michael Richards     Role: Cosmo Kramer
```

# For Loops - Iterating Dictionaries



```python
for key in cast:
    print(key)
```

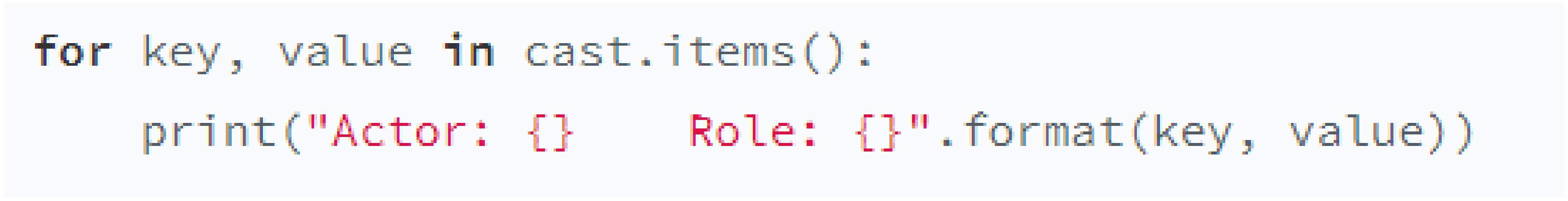


This outputs:

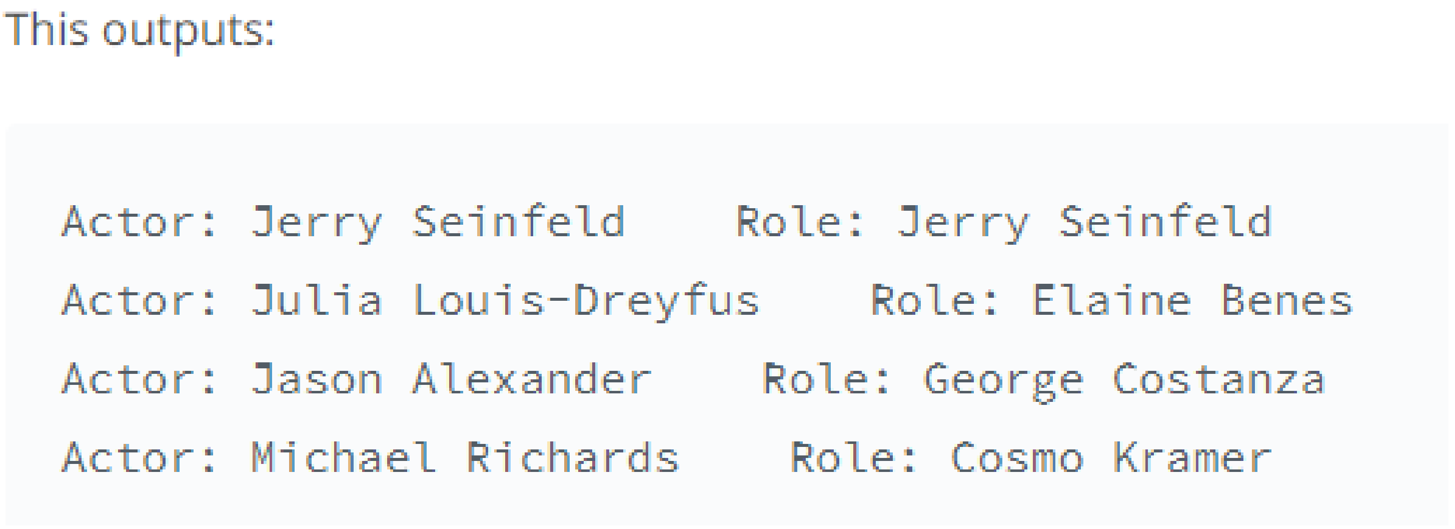
```
Jerry Seinfeld
Julia Louis-Dreyfus
Jason Alexander
Michael Richards
```

## For Loops - Iterate through both keys and values

## Iterate through both keys and values



```python
for key, value in cast.items():
    print("Actor: {}    Role: {}".format(key, value))
```



This outputs:

```
Actor: Jerry Seinfeld    Role: Jerry Seinfeld
Actor: Julia Louis-Dreyfus    Role: Elaine Benes
Actor: Jason Alexander    Role: George Costanza
Actor: Michael Richards    Role: Cosmo Kramer
```

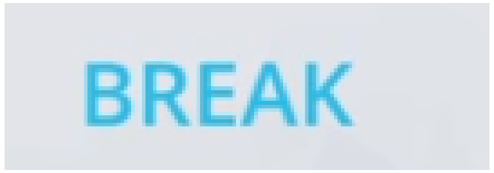
## While Loops

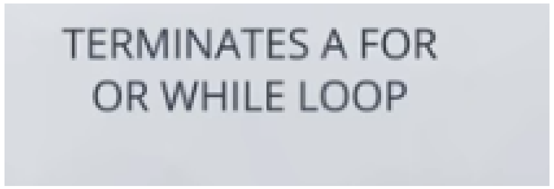
## While Loops

**Use case**:

When the company has limited budget and it decides to give hike to all candidates till the hike reaches **100000$** and to stop post that

## Break and Continue


BREAK

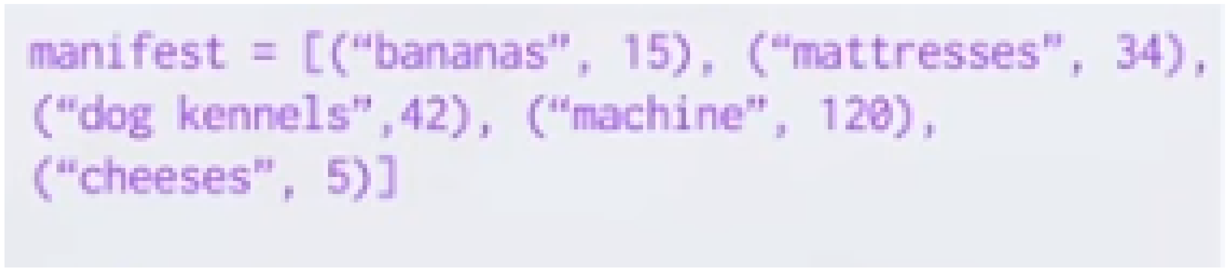

TERMINATES A FOR
OR WHILE LOOP

# Break

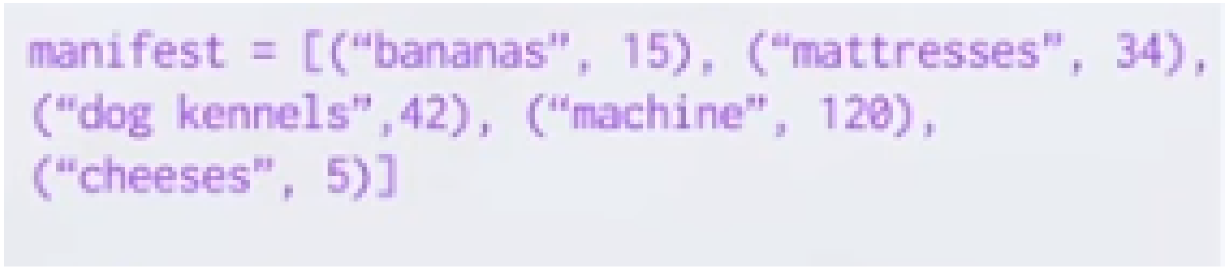
**Scenario:**

**You need to load a cargo ship with different items in the list. But the ship can load atmost 100 tonnes and the loading must be stopped if the total weight exceeds so.**

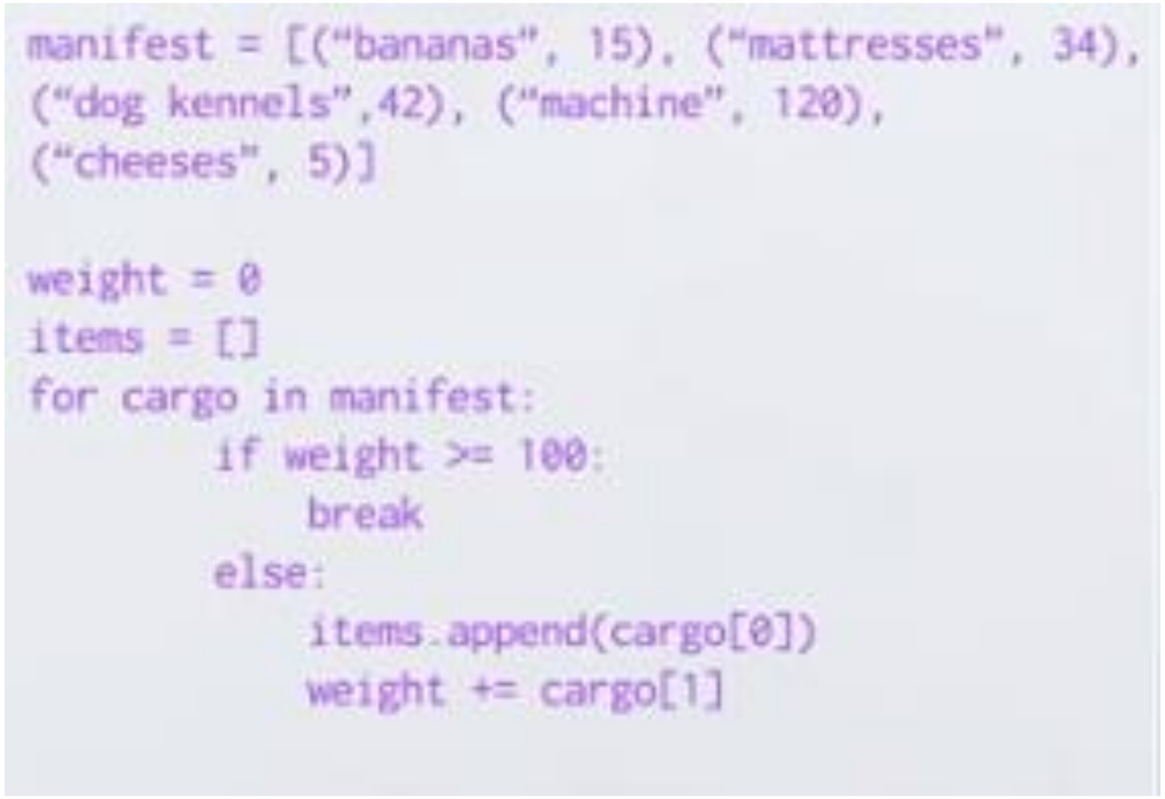
**What all the items will get loaded into the ship ?**



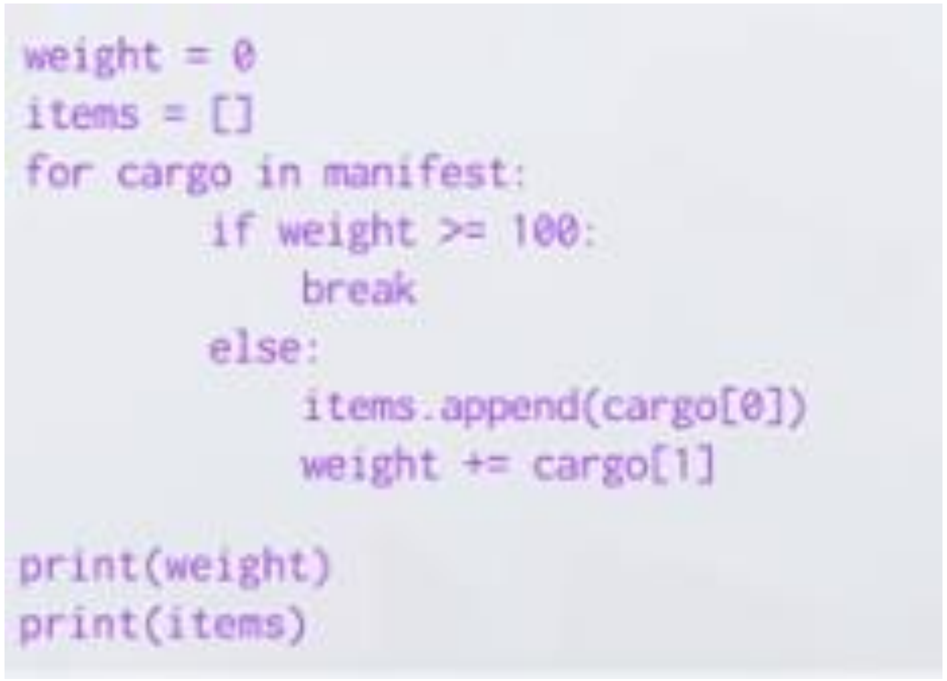
```
manifest = [("bananas", 15), ("mattresses", 34),
("dog kennels",42), ("machine", 120),
("cheeses", 5)]
```



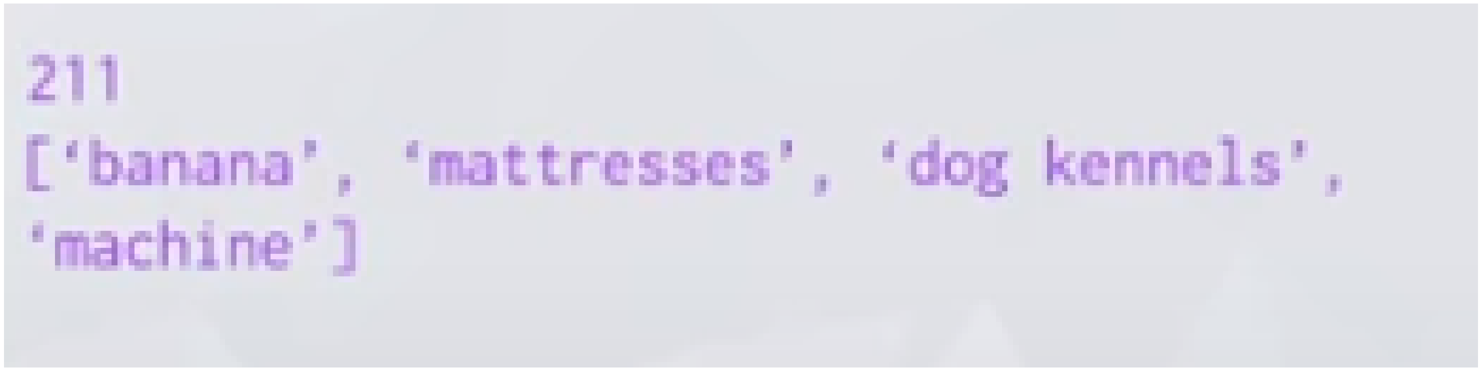
```
manifest = [(“bananas”, 15), (“mattresses”, 34),
(“dog kennels”,42), (“machine”, 120),
(“cheeses”, 5)]
```



```
manifest = [(“bananas”, 15), (“mattresses”, 34),
(“dog kennels”,42), (“machine”, 120),
(“cheeses”, 5)]

weight = 0
items = []
for cargo in manifest:
        if weight >= 100:
            break
        else:
            items.append(cargo[0])
            weight += cargo[1]
```



```
weight = 0
items = []
for cargo in manifest:
        if weight >= 100:
            break
        else:
            items.append(cargo[0])
            weight += cargo[1]

print(weight)
print(items)
```



```
211
['banana', 'mattresses', 'dog kennels',
'machine']
```

## CONTINUE

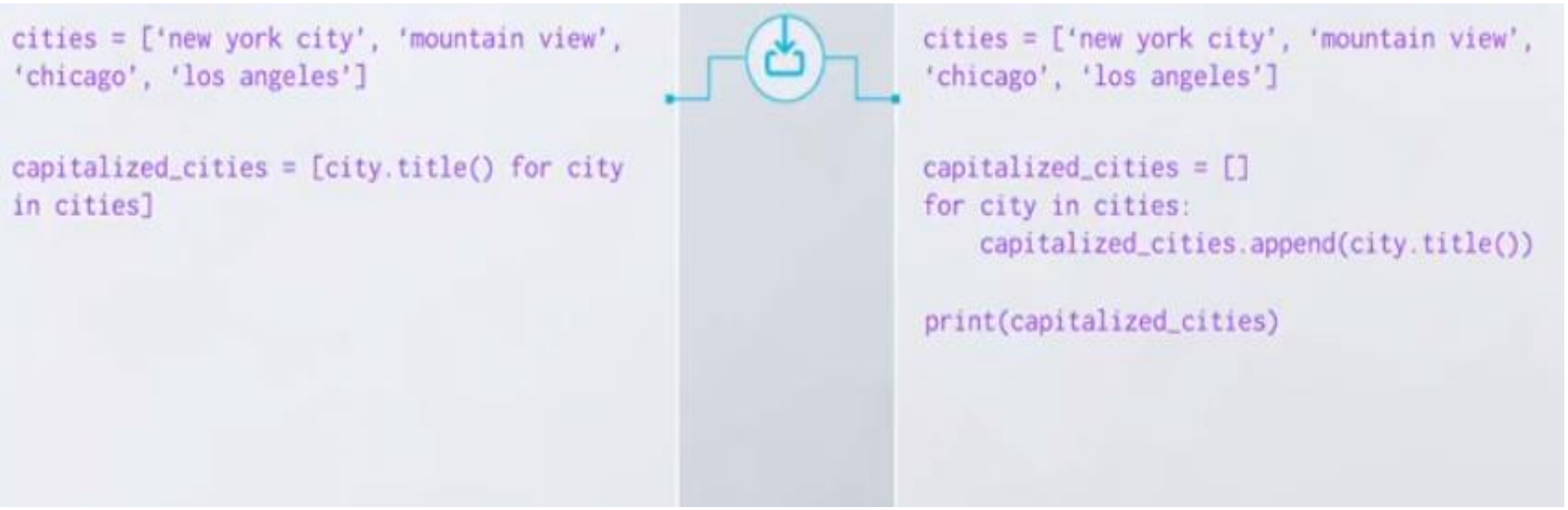
TERMINATES ONE ITERATION OF A FOR OR WHILE LOOP

# List Comprehensions

```
cities = [‘new york city’, ‘mountain view’,
‘chicago’, ‘los angeles’]
```

```
capitalized_cities = []
for city in cities:
    capitalized_cities.append(city.title())
```

```
capitalized_cities = [city.title() for city in cities]
```



```
cities = ['new york city', 'mountain view',
'chicago', 'los angeles']

capitalized_cities = [city.title() for city
in cities]
```

```
cities = ['new york city', 'mountain view',
'chicago', 'los angeles']

capitalized_cities = []
for city in cities:
    capitalized_cities.append(city.title())

print(capitalized_cities)
```

# Conditionals in List Comprehensions

## Only if -

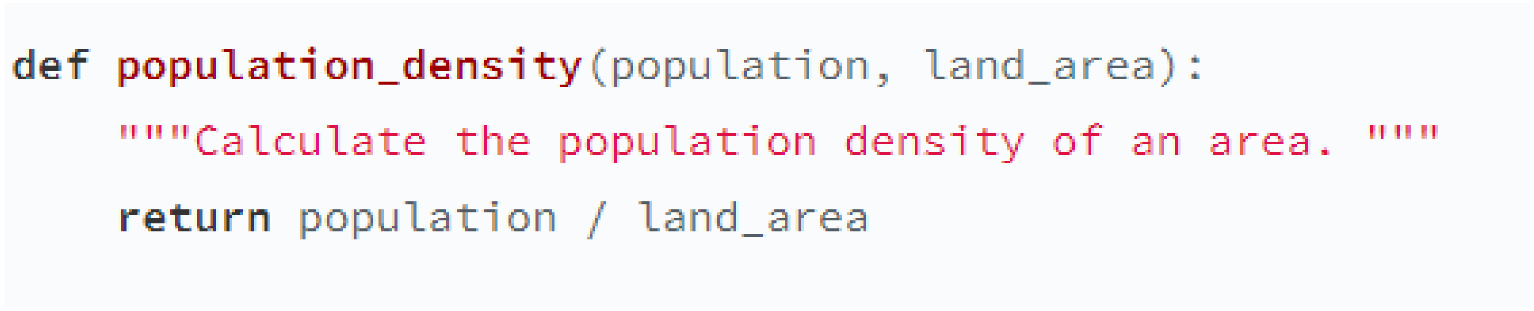
```
squares = [x**2 for x in range(9) if x % 2 == 0]
```

## With if else -

```
squares = [x**2 if x % 2 == 0 else x + 3 for x in range(9)]
```

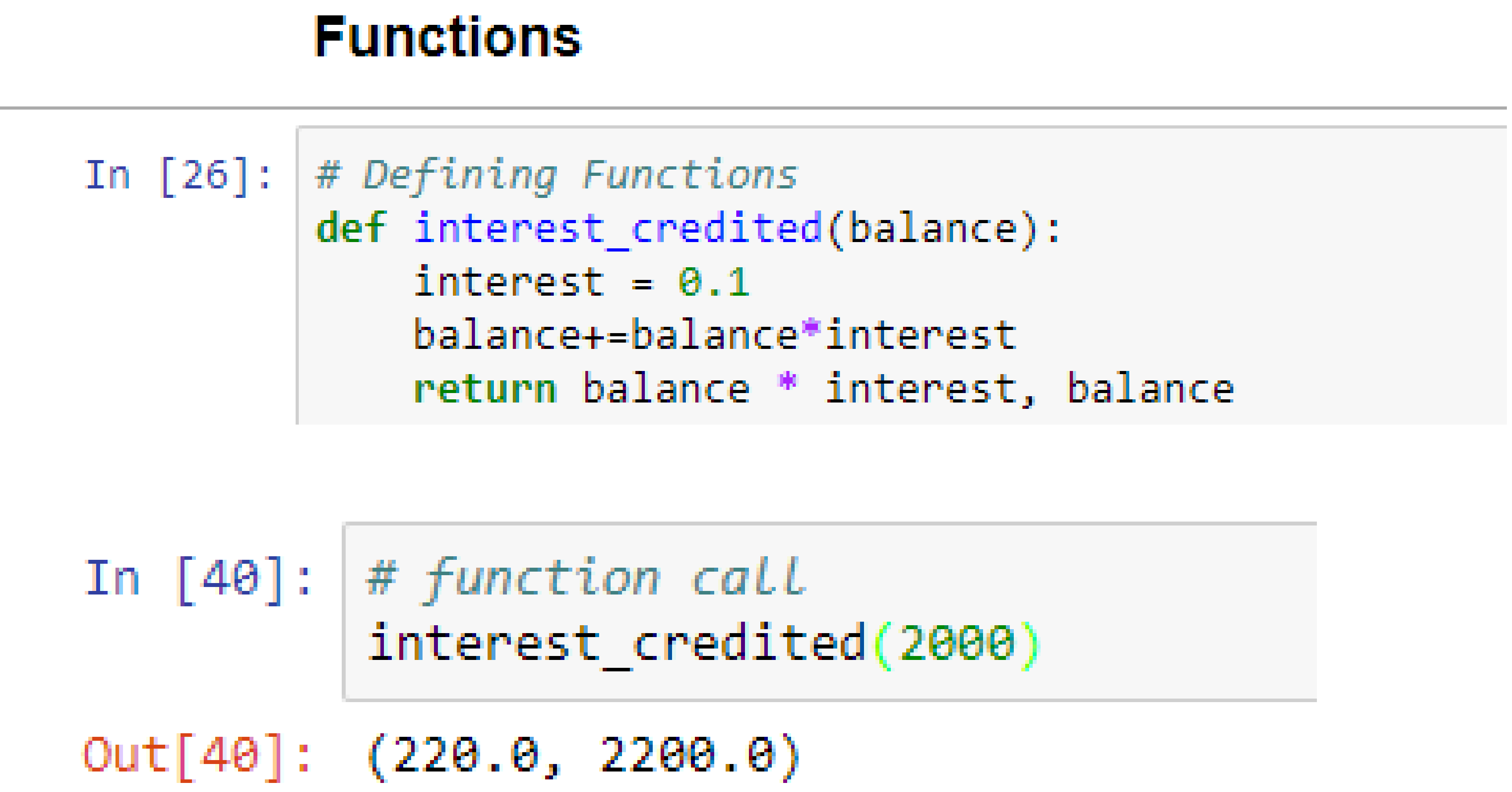
## Functions

## Functions



```python
def population_density(population, land_area):
    """Calculate the population density of an area. """
    return population / land_area
```

# Defining Functions



## Functions

In [26]:
```python
# Defining Functions
def interest_credited(balance):
    interest = 0.1
    balance+=balance*interest
    return balance * interest, balance
```

In [40]:
```python
# function call
interest_credited(2000)
```

Out[40]:
```
(220.0, 2200.0)
```