NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

EE0005 Introduction to Data Science and Artificial Intelligence

Introductory Lecture on Python Programming Language

Wonkeun Chang
wonkeun.chang@ntu.edu.sg

Outline

1.
Introduction to Python Programming Language

1.1. Python Programming Language

- Python is one of the most popular programming languages

Rank	Language	Share	Trend
1	Python	25.4%	+5.2%
2	Java	21.6%	−1.1%
3	JavaScript	8.4%	+0.0%

PYPL: PopularitY of Programming Language (pypl.github.io as of December 2018)

1.1. Python Programming Language

- Python is one of the most popular programming languages

Rank	Language	Share	Trend
1	Python	25.4%	+5.2%
2	Java	21.6%	−1.1%
3	JavaScript	8.4%	+0.0%

PYPL: PopularitY of Programming Language (pypl.github.io as of December 2018)

- Built by Guido van Rossum in 1991
- Latest version – Python 3.7
- Widely used in developing desktop/web applications and machine learning tools

Python logo

Guido van Rossum

Python logo image source: https://commons.Wikipedia.org/wiki/File:Python-logo-notext.svg

Guido van Rossum portrait image source: https://commons.Wikimedia.org/wiki/File:Guido-portrait-2014.jpg

1.2. What Makes Python Popular?

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- Programmer-friendly language – Human readable!

```
availability = {'Monday', 'Wednesday', 'Friday'}

scheduled = False
while not scheduled:
    request = input('When should we meet?')
    if request in availability:
        scheduled = True
        print('Great! I am available that day.')
    else:
        print('I am not available that day.')
```

Can you guess what it does?

1.2. What Makes Python Popular?

- Programmer-friendly language – Human readable!

```
availability = {'Monday', 'Wednesday', 'Friday'}

scheduled = False
while not scheduled:
    request = input('When should we meet?')
    if request in availability:
        scheduled = True
        print('Great! I am available that day.')
    else:
        print('I am not available that day.')
```

Can you guess what it does?

- Open source, general-purpose language
  - Over 125,000 third-party packages – A versatile tool for various tasks (see e.g., https://pypi.org)
  - Data focused packages: *pandas*, *numpy*, *matplotlib* for processing, manipulating and visualising data

1.3.1. C vs Python – Compiled vs Interpreted (1)

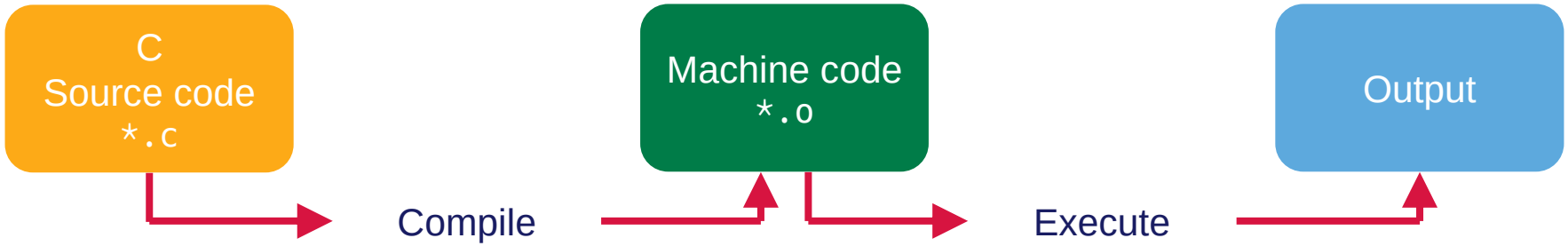
- C – Compiled

  - Convert source code to machine understandable code

C logo image source: https://commons.Wikimedia.org/wiki/File:The_C_Programming_Language_log.svg

1.3.1. C vs Python – Compiled vs Interpreted (1)

- C – Compiled



- Convert source code to machine understandable code
- Execute the machine code

! Faster execution BUT strict structure !

C logo image source: https://commons.Wikimedia.org/wiki/File:The_C_Programming_Language_log.svg

1.3.1. C vs Python – Compiled vs Interpreted (2)

- Python – Interpreted

- Execute programme directly from source code
- Source is compiled to a bytecode (`*.pyc`) during Python execution but it does NOT require an explicit compilation

! Flexible and interactive BUT slower execution !

Python logo image source: https://commons.Wikipedia.org/wiki/File:Python-logo-notext.svg

1.3.2. C vs Python – POP or OOP (1)

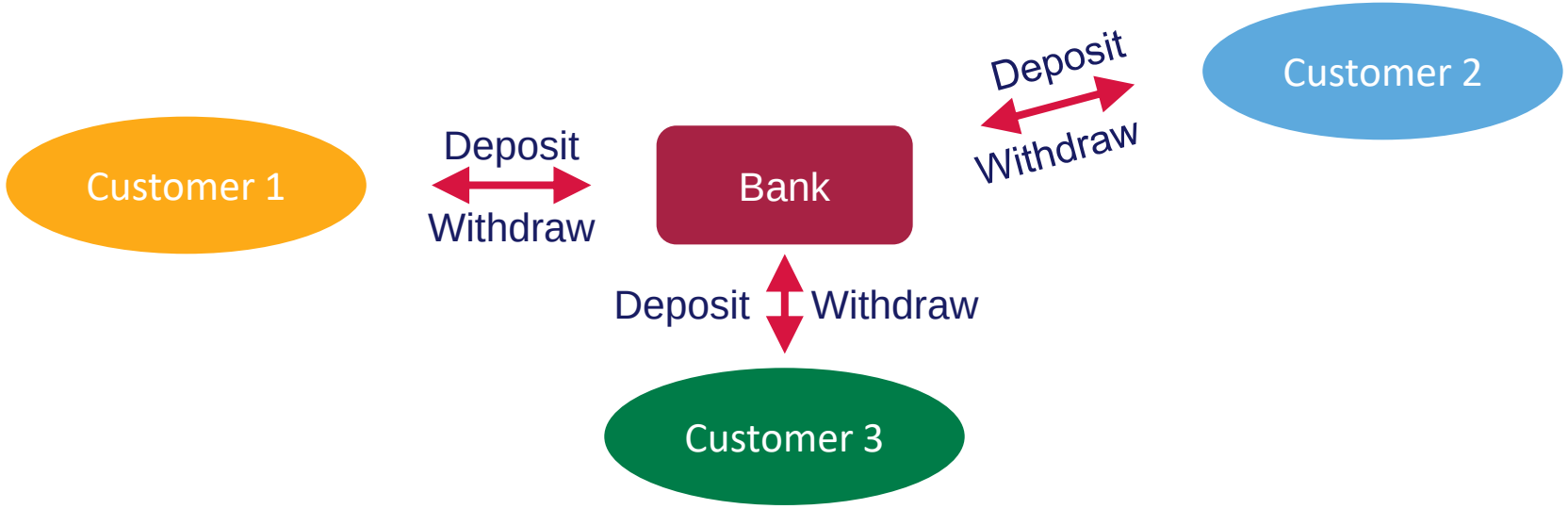
- C – Procedure oriented programming (POP) language

  - Procedures for getting tasks done in a sequential manner
  - Process focused
  - Suitable for simple tasks

C logo image source: https://commons.Wikimedia.org/wiki/File:The_C_Programming_Language_log.svg

1.3.2. C vs Python – POP or OOP (2)

- Python – Object oriented programming (OOP) language



  - Objects carrying member attributes and functions
  - Data focused
  - Suitable for complex tasks

Python logo image source: https://commons.Wikipedia.org/wiki/File:Python-logo-notext.svg

1.3.3. C vs Python – Syntax#

#Language grammar

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- C and Python have different syntax

THE C PROGRAMMING LANGUAGE

```c
#include <stdio.h>
int main()
{
    printf("Hello, World!\n");
    return 0;
}
```

```python
print('Hello, World!')
```

! You will learn Python syntax in this lecture !

C logo image source: https://commons.Wikimedia.org/wiki/File:The_C_Programming_Language_log.svg

Python logo image source: https://commons.Wikipedia.org/wiki/File:Python-logo-notext.svg

1.4. Indentation

- Define a code block

```
mark = 40
if mark<50:
    print('You have failed the test.')
else:
    print('You have passed the test.')
    print('Congratulations!')
```

```
You have failed the test.
```

```
mark = 40
if mark<50:
    print('You have failed the test.')
else:
    print('You have passed the test.')
print('Congratulations!')
```

```
You have failed the test.
Congratulations!
```

1.4. Indentation

- Define a code block

```
mark = 40
if mark<50:
    print('You have failed the test.')
else:
    print('You have passed the test.')
    print('Congratulations!')
```

```
You have failed the test.
```

Inside `else` block

```
mark = 40
if mark<50:
    print('You have failed the test.')
else:
    print('You have passed the test.')
print('Congratulations!')
```

```
You have failed the test.
Congratulations!
```

Outside `else` block

! In Python, indentation is a requirement, not a matter of style !

1.5. Comments

- Use comments to annotate codes
- Comments are ignored by Python interpreter
- Single-line comment: `# comments`

```
print('This will be excuted.') # This will not.
```

- Multi-line comment: `""" comments """`

```
"""
Write a long commment over multiple lines
in a block enclosed by a pair of three
double quotation marks.
"""
```

2.
Setting Up Python Development Environment

2.1. Python Virtual Environment and Distribution

- Set up a separate virtual environment for each project

Project A requirement

Virtual Environment A
numpy-1.15.4
h5py-2.7.0
matplotlib-3.0.2

Virtual Environment B
numpy-1.15.4
scipy-1.1.0
matplotlib-2.2.3

Project B requirement

2.1. Python Virtual Environment and Distribution

- Set up a separate virtual environment for each project

! Different versions of the same package allowed !

2.1. Python Virtual Environment and Distribution

- Set up a separate virtual environment for each project

! Different versions of the same package allowed !

- *Anaconda* – Python distribution with focus on scientific computing
  - Choose from any versions of Python when creating a virtual environment
  - Easily manage multiple virtual environments and third-party packages

2.2.1. *Anaconda* – Installation

- Install *Anaconda* for your operating system

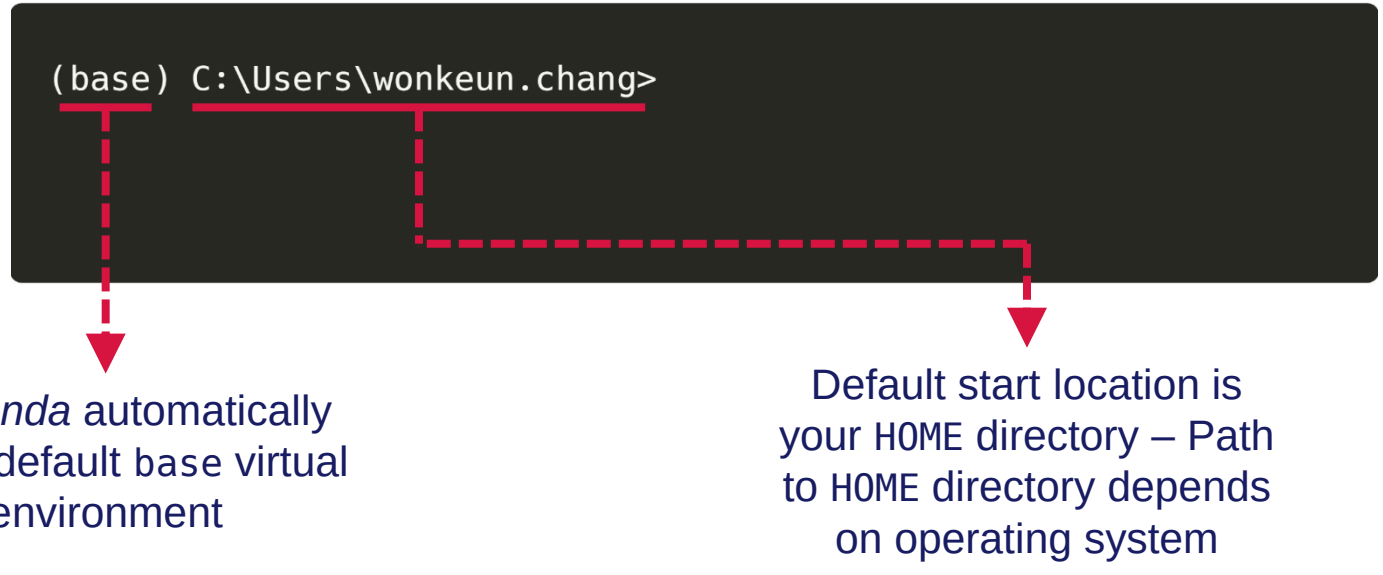
https://www.anaconda.com/download/

2.2.1. *Anaconda* – Installation

- Install *Anaconda* for your operating system

  https://www.anaconda.com/download/

- Open *Anaconda Prompt* to access Python development environment



2.2.2. *Anaconda* – Managing the Virtual Environment (1)

- Create a new virtual environment

```
(base) C:\Users\wonkeun.chang>conda create -n ee0005 python=3.6
```

Create a new virtual environment called `ee0005` with Python 3.6

Name of the new virtual environment

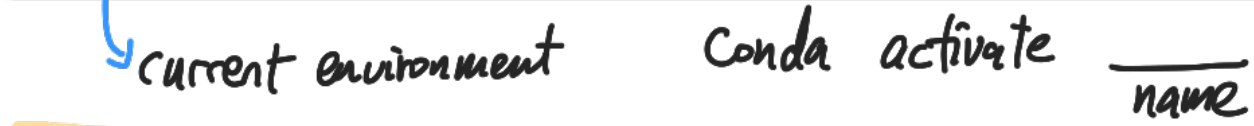
Python version for the new virtual environment – omit the version number to use the latest version, i.e. `conda create -n ee0005 python`

2.2.2. *Anaconda* – Managing the Virtual Environment (2)

- Enter an existing virtual environment

```
(base) C:\Users\wonkeun.chang>conda activate ee0005
(ee0005) C:\Users\wonkeun.chang>
```

Enter `ee0005` virtual environment



- Exit current virtual environment

```
(ee0005) C:\Users\wonkeun.chang>conda deactivate
(base) C:\Users\wonkeun.chang>
```

Exit `ee0005` virtual environment and return to `base` virtual environment

2.2.2. *Anaconda* – Managing the Virtual Environment (3)

- Manage current virtual environment

```
(ee0005) C:\Users\wonkeun.chang>conda install scipy

(ee0005) C:\Users\wonkeun.chang>conda install matplotlib=2.2.3

(ee0005) C:\Users\wonkeun.chang>conda remove scipy

(ee0005) C:\Users\wonkeun.chang>conda --help
```

Install the latest version of *scipy* package in `ee0005` virtual environment

Install *matplotlib* package version 2.2.3 in `ee0005` virtual environment

Remove *scipy* package from `ee0005` virtual environment

Print the help message

2.2.2. *Anaconda* – Managing the Virtual Environment (4)

- Delete an existing virtual environment

```
(base) C:\Users\wonkeun.chang>conda remove -n ee0005 --all
```

Completely delete `ee0005` virtual environment

Make sure to delete it from outside `ee0005` virtual environment

2.2.3. *Anaconda* – Running a Python Programme

- Create a source code file using a text editor

```
print('Hello, World!')
```

Create `hello.py` file in a directory that you can easily access

! Features: Find and Replace, Syntax Highlighting, Line Numbers, …
E.g., *Visual Studio Code*, *Atom*, *Notepad++*, *Vim*, … !

2.2.3. *Anaconda* – Running a Python Programme

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- Create a source code file using a text editor

```
print('Hello, World!')
```

Create `hello.py` file in a directory that you can easily access

- Open *Anaconda Prompt*, enter a virtual environment (if needed) and navigate to the directory with the source code

```
(ee0005) C:\Users\wonkeun.chang>cd Examples
(ee0005) C:\Users\wonkeun.chang\Examples>
```

See your operating system manual on how to navigate through directories

cd: change directory

2.2.3. *Anaconda* – Running a Python Programme

- Create a source code file using a text editor

```
print('Hello, World!')
```

Create `hello.py` file in a directory that you can easily access

- Open *Anaconda Prompt*, enter a virtual environment (if needed) and navigate to the directory with the source code

```
(ee0005) C:\Users\wonkeun.chang>cd Examples
(ee0005) C:\Users\wonkeun.chang\Examples>
```

See your operating system manual on how to navigate through directories

- Run the Python programme


```
(ee0005) C:\Users\wonkeun.chang\Examples>python hello.py
Hello, World!
```

Enter `python hello.py` to execute

3.
Variables and Data Types

3.1. Variables

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

" " (array)

str, int, list, float, bool, tuple

- Created when assigned with a value
  - Use = to assign a value to a variable
  - No need to declare Data Type – Automatically detected

```
>>> a = 'hi'                                    # str
>>>
>>> b = 1+10                                    # int
>>>
>>> c = [a, ' there']                           # list
>>>
>>> d = b+0.5                                   # float
>>>
>>> e = b==d                                    # bool
>>>
>>> (a, b, c, d, e)                             # tuple
('hi', 11, ['hi', ' there'], 11.5, False)
>>>
>>> type(a), type(b), type(c), type(d), type(e) # tuple
(<class 'str'>, <class 'int'>, <class 'list'>, <class 'float'>, <class 'bool'>)
```

type(a) ↓ checks data type

3.2. Classes and Objects (1)

"variables" in objects

- Object – A collection of attributes and functions that act on attributes
- Class – A template for an object

! An object is an instance of a class

Attributes and functions of an object are defined in its class !

3.2. Classes and Objects (2)

- E.g. Keep a record of sales employees
  - Create a class – Template

__init__ → function objects when defining

SalesEmployee

Attributes: name, salary, sales
Functions: deal_closed, raise_salary

```
class SalesEmployee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        self.sales = 0.0
    def deal_closed(self, amount):
        self.sales += amount
    def raise_salary(self, percent):
        self.salary *= 1+percent/100
```

  - Create objects – Instances of the class

amy

name: 'Amy Alton'
salary: 3200.00
sales: 0.00

ben

name: 'Ben Brown'
salary: 2950.00
sales: 0.00

```
amy = SalesEmployee('Amy Alton', 3200.00)
ben = SalesEmployee('Ben Brown', 2950.00)
```

3.2. Classes and Objects (3)

- Operate on objects through their member functions

amy

```
name: 'Amy Alton'
salary: 3200.00
sales: 120.00
```

ben

```
name: 'Ben Brown'
salary: 3097.50
sales: 0.00
```

```
amy.deal_closed(120.00)
ben.raise_salary(5)
```

- Access object member attributes

amy

```
name: 'Amy Alton'
salary: 3200.00
sales: 120.00
```

ben

```
name: 'Ben Brown'
salary: 3097.50
sales: 0.00
```

```
print(amy.name+"'s total sales figures:", amy.sales)
print(ben.name+"'s current salary:", ben.salary)
```

```
Amy Alton's total sales figure: 120.0
Ben Brown's current salary: 3097.5
```

3.2. Classes and Objects (4)

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- In Python, EVERY variable is an object belonging to a class

! Python has many handy “built-in” classes

They are called Data Types (historical reason) !

3.3. Standard Data Types (1)

- Integer: `int`

```
n = 105
```

```
level = -2
```

- Floating point: `float`

```
pi = 3.14159
```

```
e = -1.602e-19
```

- Complex: `complex`

$j = \sqrt{-1}$

```
z = -1+3.2j
```

```
i = (-1)**(0.5)
```

! `int`, `float`, `complex` are Numeric Data Types !

3.3. Standard Data Types (2)

- String: `str`

```
university = 'NTU'
```

```
s = "I'm a string!"
```

  - Enclosed by a pair of single (`'string'`) or double (`"string"`) quotation marks

- Boolean: `bool`

```
weekend = True
```

```
expired = False
```

  - Two possible values: `True`, `False`

3.3. Standard Data Types (3)

- List: `list` [,]

```
year = [2018, 2019, 2020]
```

```
mixed = [0, 2.72, 'Mon']
```

  - Ordered – Access individual items using indices
  - Mixed Data Type allowed

0-based

- Tuple: `tuple` → list but can't change it's data

(,) optional

```
primary_colours = ('red', 'green', 'blue')
```

  - Mostly same as `list`
  - Difference: reassignment or deletion of individual items NOT allowed

3.3. Standard Data Types (4)

- Dictionary: `dict` { _ : _ , … }

```
car = {'brand': 'Honda', 'model': 'Civic', 'price': 99000.0}
```

  - Hold key-value pairs
  - Unordered – Access values using keys

- Set: `set` { , }

```
fruits = {'apple', 'banana', 'cherry'}
```

  - No duplicate values – Individual items are unique
  - Unordered

3.4.1. Basic Operations – `int`, `float`, `complex` (1)

- Arithmetic
  - Basic arithmetic: +, –, *, /

```
>>> 1-5, 2/5, 1.2e3*-3, (3+4j)+2
(-4, 0.4, -3600.0, (5+4j))
```

  - Floor division: //

```
>>> 5//2, -4.0//3, 2.128/127
(2, -2.0, 0.0)
```

  - Exponent: **

```
>>> 3**2, 2**(0.5), 10**-1, (2-1j)**2
(9, 1.4142135623730951, 0.1, (3-4j))
```

! Return `int`, `float`, `complex` !

3.4.1. Basic Operations – `int`, `float`, `complex` (2)

- Comparison
  - Equal, not equal: `==`, `!=` → avoid on floats

```
>>> 2+1==5-2, 2!=-2, 1+(4/3)**2==(5/3)**2
(True, True, False)
```

  - Greater, greater or equal, less, less or equal: `>`, `>=`, `<`, `<=` #

```
>>> -3.5>2, 7//3<=2, -4>2-6
(False, True, False)
```

#Cannot operate on `complex`

! Return `bool` !

3.4.2. Basic Operations – bool (1)

- Logical
  - not

```
>>> not False, not 2!=-2, not -3.5>2
(True, False, True)
```

  - or

```
>>> False or True, 2==3 or 4-1==1-4
(True, False)
```

  - and

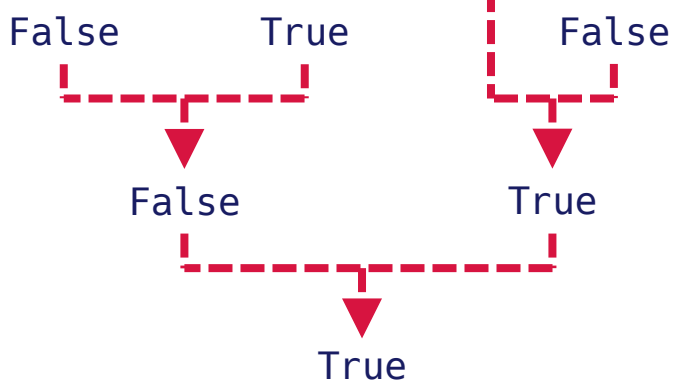
```
>>> False and True, 2==3-1 and 2*-1==-2
(False, True)
```

! Return bool !

3.4.2. Basic Operations – bool (2)

- Logical combinations

```
>>> 2*6<=10 and 32/8>=4 or not 5**2<25
True
```



order: ① value of bool
② NOT
③ AND/OR left → right

- Arithmetic/comparison on `bool`
  - `True` is equivalent to `int(1)`
  - `False` is equivalent to `int(0)`

```
>>> True+True+True, False*True, True/2
(3, 0, 0.5)
```

3.4.3. Basic Operations – `str` (1)

- Concatenate: `+` s1+s2

```
>>> s1, s2 = 'Hello, ', 'World!'
>>> s1+s2
'Hello, World!'
```

! Return `str` !

- Get length: `len` len( )

```
>>> s = 'Hello, World! Hello, Python!'
>>> len(s)
28
```

! Return `int` !

- Count a substring occurrence: `count`

```
>>> s = 'Hello, World! Hello, Friends!'
>>> s.count('o'), s.count('Hello')
(3, 2)
```

! Return `int` !

str.count(_)
↳ substring

3.4.3. Basic Operations – `str` (2)

- Locate a substring: `index`

```
>>> s = 'mozzareLla'
>>> s.index('z'), s.index('l'), s.index('are')
(2, 8, 4)
```

str.index(__)
↳ substring

! Return `int` !

- Use zero-based indexing

index	0	1	2	3	4	5	6	7	8	9
s	m	o	z	z	a	r	e	L	l	a

- Find index of the first occurrence of the first letter in substring

! All `str` operations are cASe senSitivE !

3.4.3. Basic Operations – `str` (3)

- Slice: `[start:stop:step]`

```
>>> s = 'mozzareLla'
>>> s[2], s[2:9], s[2:9:3], s[2:-2], s[-2:1:-2]
('z', 'zzareLl', 'zrl', 'zzareL', 'leaz')
```

index: 0 ~ 9

neg. index: -10 ~ -1

! Return `str` !

- From `start` up to but NOT including `stop` picking item every `step`
- Negative index – Alternative index that runs backward from the last character
- Negative `step` – Slice in reverse direction

index	0	1	2	3	4	5	6	7	8	9
s	m	o	z	z	a	r	e	L	l	a
negative index	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

3.4.3. Basic Operations – `str` (4)

- Shorthand notations

```
>>> s = 'mozzareLla'
>>> s[::-1]   # all characters, reversed
'alLerazzom'
>>> s[-3:]    # last three characters
'Lla'
>>> s[:-2]    # all except last two characters
'mozzareL'
>>> s[1::-1]  # first two characters, reversed
'om'
>>> s[:-3:-1] # last two characters, reversed
'al'
```

omitted:
start - beginning
stop - end
step - forward 1

`start`, `stop`, `step` in shorthand notations are determined by their positions with respect to `:`

index	0	1	2	3	4	5	6	7	8	9
s	m	o	z	z	a	r	e	L	l	a
negative index	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

3.4.3. Basic Operations – `str` (5)

- Cast to upper/lowercase: `upper/lower`

```
>>> s = 'Hello, World!'
>>> s.upper(), s.lower()
('HELLO, WORLD!', 'hello, world!')
```

str.upper()
↳ changes the string

! Return `str` !

- Replace a substring: `replace`

str.replace(str1, str2)

multiple times:

```
>>> s = 'I met Amy on Monday. Do you know Amy?'
>>> s.replace('Amy', 'Ben')
'I met Ben on Monday. Do you know Ben?'
>>>
>>> s.replace('Amy', 'Ben').replace('Monday', 'Friday')
'I met Ben on Friday. Do you know Ben?'
```

also string

str.replace(...).replace(...)

`str`

! Return `str` !

3.4.3. Basic Operations – `str` (6)

- Break `str` at separator: `split`

```
>>> s = 'Hello, World! Hello, Friends!'
>>> s.split()
['Hello,', 'World!', 'Hello,', 'Friends!']
>>>
>>> s.split(',')
['Hello', ' World! Hello', ' Friends!']
>>>
>>> s.split('Hello, ')
['', 'World! ', 'Friends!']
```

str.split( )
↓
Seperator
(does not appear in list)

! Return `list` !

- Default separator – whitespace
- Useful for importing, e.g. CSV (comma-separated values) file

H	e	l	l	o	,		W	o	r	l	d	!		H	e	l	l	o	,		F	r	i	e	n	d	s	!

3.4.4. Basic Operations – `list` (1)

- Ordered collection of data
- Use indices to access, reassign or delete items

```
>>> rainfall = [0, 3.2, 15.6, 0.2, 0, 0, 43.8, 0]
>>> rainfall[2]       # access third item
15.6
>>>
>>> rainfall[4] = 0.2 # reassign fifth item
>>> rainfall
[0, 3.2, 15.6, 0.2, 0.2, 0, 43.8, 0]
>>>
>>> del rainfall[-1]  # delete last item
>>> rainfall
[0, 3.2, 15.6, 0.2, 0.2, 0, 43.8]
```

del list[i] → delete index i
(positive/negative index)

3.4.4. Basic Operations – `list` (2)

- Many basic `str` operations apply

```
>>> rainfall = [0, 3.2, 15.6, 0.2, 0.2, 0, 43.8]
>>> rainfall += [0, 0, 0]
>>> rainfall
[0, 3.2, 15.6, 0.2, 0.2, 0, 43.8, 0, 0, 0]
>>>
>>> len(rainfall)
10
>>> rainfall.count(0)
5
```

+= : concentate data behind

len(list)

list.count(_)

! Return `int` !

index	0	1	2	3	4	5	6	7	8	9
rainfall	0	3.2	15.6	0.2	0.2	0	43.8	0	0	0

3.4.4. Basic Operations – `list` (3)

- Slice `list` in the same way as `str`

new list itself

↑

list[: :]

del list[: :]

↓

updates list

list[: :] = [...]

↳ reassigns original list

```
>>> rainfall = [0, 3.2, 15.6, 0.2, 0.2, 0, 43.8]
>>> rainfall[2:5]                # from third to fifth items
[15.6, 0.2, 0.2]
>>> rainfall[::-1]               # all items, reversed
[43.8, 0, 0.2, 0.2, 15.6, 3.2, 0]
>>> rainfall[:-2]                # all except last two items
[0, 3.2, 15.6, 0.2, 0.2]
>>> rainfall[-2:] = [0.4, 6.2] # reassign last two items
>>> rainfall
[0, 3.2, 15.6, 0.2, 0.2, 0.4, 6.2]
>>> del rainfall[-3:]            # delete last three items
>>> rainfall
[0, 3.2, 15.6, 0.2]
```

} doesn't manipulate original list

→ manipulates original list

index	0	1	2	3	4	5	6
rainfall	0	3.2	15.6	0.2	0.2	0.4	6.2
negative index	-7	-6	-5	-4	-3	-2	-1

3.4.4. Basic Operations – `list` (4)

- Add items: `append`, `insert`

```
>>> rainfall = [0, 3.2, 0.2, 0.2, 0]
>>> rainfall.append(43.8)
>>> rainfall
[0, 3.2, 0.2, 0.2, 0, 43.8]
>>>
>>> rainfall.insert(2, 15.6)
>>> rainfall
[0, 3.2, 15.6, 0.2, 0.2, 0, 43.8]
```

list.append(_): add to back

list.insert(i, _) insert new item as index i

indices shift ►►►

index	0	1	2	3	4	5	6
rainfall	0	3.2	15.6	0.2	0.2	0	43.8

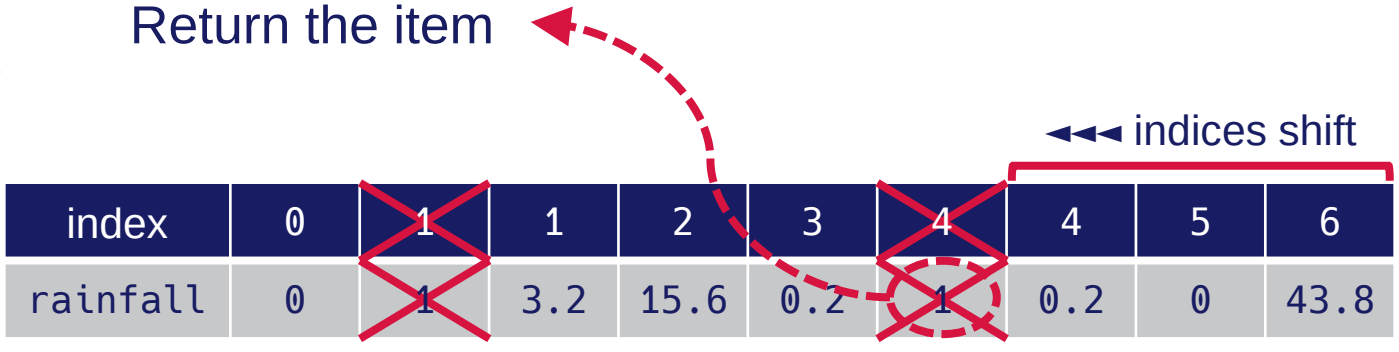
3.4.4. Basic Operations – `list` (5)

- Remove items: `remove`, `pop`

list.remove(_)
: removes first occurrence of _

list.pop(i)
: removes index i, also returns it

```
>>> rainfall = [0, 1, 3.2, 15.6, 0.2, 1, 0.2, 0, 43.8]
>>> rainfall.remove(1) # remove first occurrence of 1
>>> rainfall
>>> [0, 3.2, 15.6, 0.2, 1, 0.2, 0, 43.8]
>>>
>>> rainfall.pop(4)    # remove and return fifth item
1
>>> rainfall
[0, 3.2, 15.6, 0.2, 0.2, 0, 43.8]
```



3.4.4. Basic Operations – `list` (6)

- Sort items: `sort`

manipulates original list data

```
>>> rainfall = [0, 3.2, 15.6, 0.2, 0.2, 0, 43.8]
>>> rainfall.sort()
>>> rainfall
[0, 0, 0.2, 0.2, 3.2, 15.6, 43.8]
>>>
>>> rainfall.sort(reverse=True)
>>> rainfall
[43.8, 15.6, 3.2, 0.2, 0.2, 0, 0]
```

  - Default – Sort in ascending order
  - Use `reverse=True` to sort in descending order

! Operate on `list` itself and destroy the original order in `list` !

list.sort(option)

rev: reverse=True

index	0	1	2	3	4	5	6
rainfall	0	3.2	15.6	0.2	0.2	0	43.8

3.4.4. Basic Operations – `list` (7)

- Nested `list` – `list` within another `list` as its item

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8]]
>>> type(matrix), type(matrix[1]), type(matrix[1][2])
(<class 'list'>, <class 'list'>, <class 'int'>)
>>>
>>> len(matrix), len(matrix[1]), len(matrix[2])
(3, 3, 2)
```

2 items

index 1	0			1			2	
index 2	0	1	2	0	1	2	0	1
matrix	1	2	3	4	5	6	7	8

3.4.4. Basic Operations – `list` (8)

- Same slice and `list` operations apply

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8]]
>>> matrix[:2], matrix[1][-2:]
([[1, 2, 3], [4, 5, 6]], [5, 6])
>>>
>>> matrix[1].sort(reverse=True)
>>> matrix
[[1, 2, 3], [6, 5, 4], [7, 8]]
```

ALLOWED

- Nested `list` is allowed with `list`, `tuple`, `dict` or `set` as its items

! Use *numpy* arrays for handling large data !

index 1	0			1			2	
index 2	0	1	2	0	1	2	0	1
matrix	1	2	3	6	5	4	7	8

3.4.5. Basic Operations – `tuple` (1)

- Ordered collection of data similar to `list`

- len()

* tuple CANNOT reassign items, but CAN append another tuple

```
>>> cities = ('Paris', 'Berlin', ('Rome', 'Vatican'))
>>> type(cities), type(cities[0]), type(cities[2])
(<class 'tuple'>, <class 'str'>, <class 'tuple'>)
>>>
>>> len(cities), len(cities[2])
(3, 2)
>>>
>>> cities[1] = 'Madrid'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

! Individual items CANNOT be changed !

3.4.5. Basic Operations – `tuple` (1)

- Ordered collection of data similar to `list`

```
>>> cities = ('Paris', 'Berlin', ('Rome', 'Vatican'))
>>> type(cities), type(cities[0]), type(cities[2])
(<class 'tuple'>, <class 'str'>, <class 'tuple'>)
>>>
>>> len(cities), len(cities[2])
(3, 2)
>>>
>>> cities[1] = 'Madrid'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

! Individual items CANNOT be changed !

- Nested `tuple` is allowed with `list`, `tuple`, `dict` or `set` as its items

3.4.5. Basic Operations – `tuple` (2)

- Many operations are same as `str` +=, slice, .count, .index

* tuple length 1:

nothing after 1st comma

```
>>> cities = 'Paris', 'Berlin', 'Rome'
>>> cities += ('Berlin', )
>>> cities
('Paris', 'Berlin', 'Rome', 'Berlin')
>>>
>>> cities[:-1]
('Paris', 'Berlin', 'Rome')
>>>
>>> cities.count('Berlin')
2
>>>
>>> cities.index('Rome')
2
```

! Return `int` !

index	0	1	2	3
cities	'Paris'	'Berlin'	'Rome'	'Berlin'

3.4.6. Basic Operations – `dict` (1)

- No index for item access – Use keys

```
>>> age = {'Claire': 5, 'Ben': 4, 'Amy': 8}
>>> age['Amy']     # access an item using its key
8
>>>
>>> age['Ben'] = 5  # update an item
>>> age
{'Claire': 5, 'Ben': 5, 'Amy': 8}
>>>
>>> age['Dave'] = 8  # add a new item
>>> age
{'Claire': 5, 'Ben': 5, 'Amy': 8, 'Dave': 8}
>>>
>>> del age['Claire'] # delete an item
>>> age
{'Ben': 5, 'Amy': 8, 'Dave': 8}
```

dict = { key : value , ... }

dict[key] = value

age

key	value
'Ben'	5
'Amy'	8
'Dave'	8

nested: tuple — tuple, list, dict, set

- Nested `dict` is allowed with only `tuple` as its keys, and `list`, `tuple`, `dict` or `set` as its values

del dict[key] ⇒ deletes item from dict

3.4.6. Basic Operations – `dict` (2)

- Get all keys/values: `keys/values`

```
>>> age = {'Claire': 5, 'Ben': 4, 'Amy': 8, 'Dave': 8}
>>> age.keys()
dict_keys(['Claire', 'Ben', 'Amy', 'Dave'])
>>>
>>> age.values()
dict_values([5, 4, 8, 8])
```

  - Return `dict_keys` – A special `list` of `dict` keys
  - Return `dict_values` – A special `list` of `dict` values

age

key	value
'Claire'	5
'Ben'	4
'Amy'	8
'Dave'	8

dict.keys() ⇒ returns list of keys } pairs have same index

! A key/value pair has the same index in `dict_keys` and `dict_values` !

dict.values() ⇒ returns list of values

3.4.7. Basic Operations – `set` (1)

- Unordered collection of unique items

```
>>> engineers = {'Amy', 'Ben', 'Claire', 'Ben'}
>>> engineers
{'Ben', 'Claire', 'Amy'}
```

  - The second occurrence of ‘`Ben`’ is not registered

- Nested `set` is allowed only with `tuple` as its items

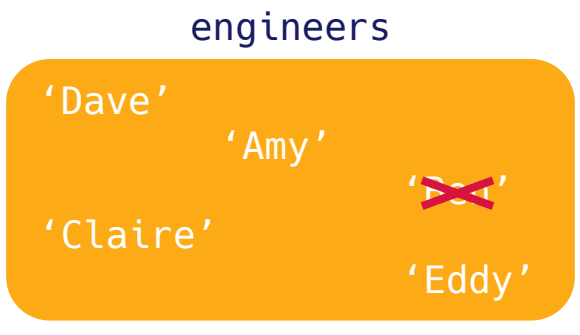
3.4.7. Basic Operations – set (2)

- Add/remove items: `update/remove` → manipulate set itself

```
>>> engineers = {'Amy', 'Ben', 'Claire'}
>>> engineers
{'Ben', 'Claire', 'Amy'}
>>>
>>> engineers.update({'Dave', 'Eddy'})
>>> engineers
{'Ben', 'Eddy', 'Dave', 'Claire', 'Amy'}
>>>
>>> engineers.remove('Ben')
>>> engineers
{'Eddy', 'Dave', 'Claire', 'Amy'}
```

set.update(__)
set/element
⇒ adds set together

set.remove(__)
set/element



3.4.7. Basic Operations – set (3)

- Union, intersection, difference, symmetric difference: |, &, -, ^

union: |

intersection: &

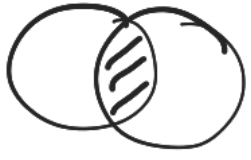
diff: -

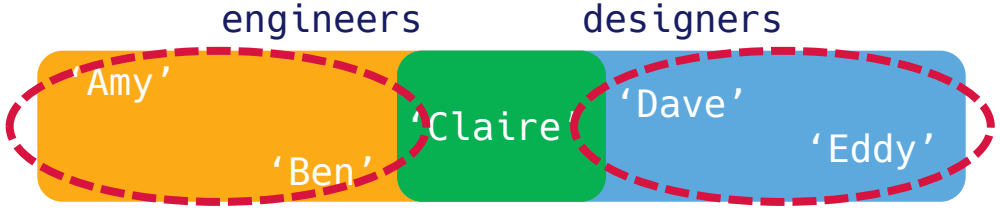
symmetric diff: ^

⇓

returns a new set

```
>>> engineers = {'Amy', 'Ben', 'Claire'}
>>> designers = {'Eddy', 'Dave', 'Claire'}
>>> engineers|designers
{'Ben', 'Eddy', 'Dave', 'Claire', 'Amy'}
>>>
>>> engineers&designers
{'Claire'}
>>>
>>> engineers-designers, designers-engineers
({'Ben', 'Amy'}, {'Dave', 'Eddy'})
>>>
>>> engineers^designers
{'Ben', 'Dave', 'Amy', 'Eddy'}
```





3.5. Data Type Conversion (1)

- Implicit – Data Type is determined automatically after an operation

```
>>> a, b, pi = 2, 4, 3.14159
>>> type(a), type(b), type(pi)
(<class 'int'>, <class 'int'>, <class 'float'>)
>>>
>>> type(a*b), type(a*pi)
(<class 'int'>, <class 'float'>)
```

`int*float` operation return `float` to avoid the loss of data

`int*int` operation return `int`

3.5. Data Type Conversion (2)

- Explicit – Data Type is converted manually: `dtype`

* assignment:

a, b, c = _, _, _

```
>>> age, s1, s2 = 21, "I'm ", " years old."
>>> type(age), type(s1), type(s2)
(<class 'int'>, <class 'str'>, <class 'str'>)
>>>
>>> s1+age+s2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>>
>>> s1+str(age)+s2
"I'm 21 years old."
```

Explicit Data Type conversion from `int` to `str`

! Also known as type casting !

int( )
str( )
⋮

3.6. Identity and Membership (1)

- Identity test: `is`, `is not`

```
>>> a, b = [1, 2, 3], [1, 2, 3]
>>> a==b, a is b, a is not b
(True, False, True)
>>>
>>> c = b
>>> c==b, c is b, c is not b
(True, True, False)
```

  - `==`, `!=` check if the values are equal
  - `is`, `is not` check if the variables point to the same object
  - `c = b` does NOT create a new object – Assign an additional variable name `c` to the same object

3.6. Identity and Membership (2)

- Membership test: `in`, `not in`

```
>>> 3 in [1, 2, 3, 4], 0 not in [1, 2, 3, 4]
(True, True)
>>>
>>> s = 'Hello, Python!'
>>> 'World' in s, 'Python' in s
(False, True)
>>>
>>> a = {'b': 1, 'c': 2, 'd': 3}
>>> 2 not in a, 'b' in a, 'e' in a
(True, True, False)
```

- Check if an object is present in a `str` or a Data Structure (`list`, `tuple`, `dict`, `set`)
- For `dict`, only the keys are checked, NOT the values

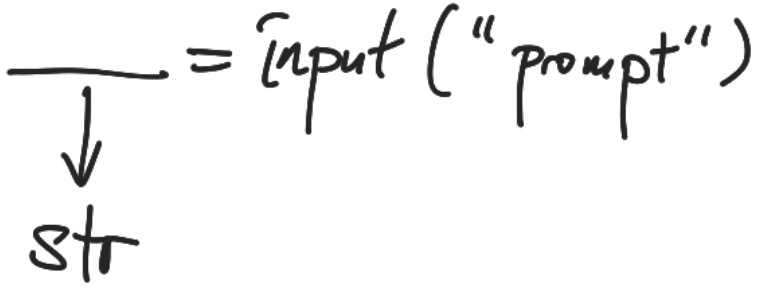
4.
Control Flow

4.1. Conditional Expressions (1)

- `if`

```
fruits = {'apple', 'orange', 'mango'}
item = input('What are you offering? ')

if item in fruits:
    print('You are offering some fruits.')

print('Thanks for the offer')
```

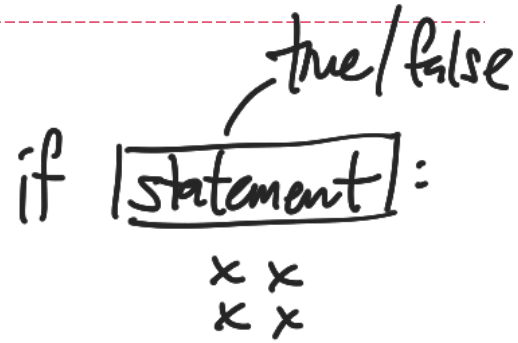
- Present a prompt message and wait for a user input
- Assign the user input to the variable as `str`



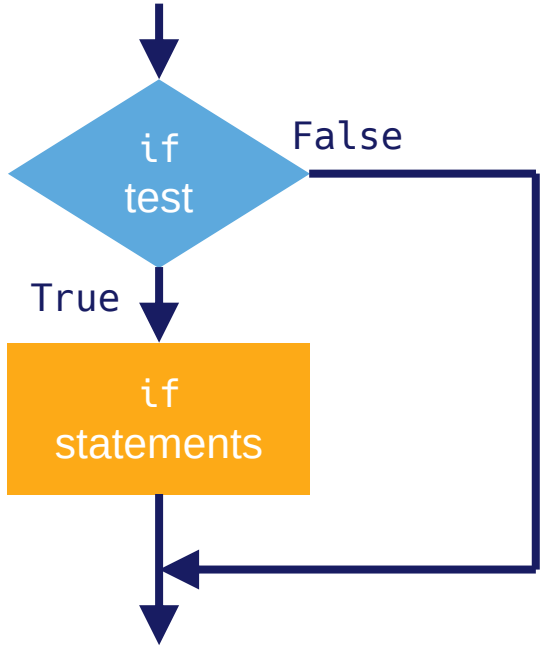
4.1. Conditional Expressions (1)

- `if`

```
fruits = {'apple', 'orange', 'mango'}
item = input('What are you offering? ')

if item in fruits:
    print('You are offering some fruits.')

print('Thanks for the offer')
```



- Test whether the `if` expression is `True` or `False`
- Execute the indented `if` statements if the expression is `True`
- Omit the indented `if` statements if the expression is `False`



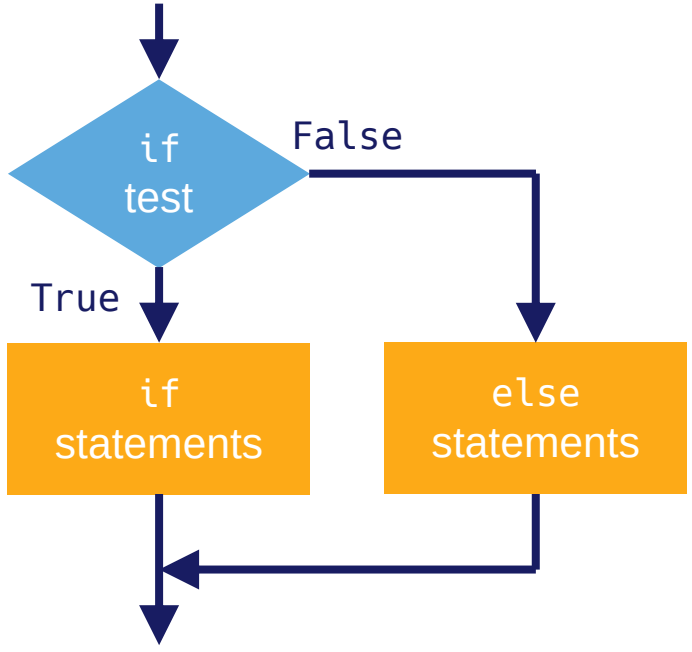
4.1. Conditional Expressions (2)

- `if, else`

```
fruits = {'apple', 'orange', 'mango'}
item = input('What are you offering? ')

if item in fruits:
    print('You are offering some fruits.')
else:
    print('You are not offering any fruits.')

print('Thanks for the offer!')
```

- Execute the indented `else` statements if `False`

- Test whether the `if` expression is `True` or `False`
- Execute the indented `if` statements if `True`



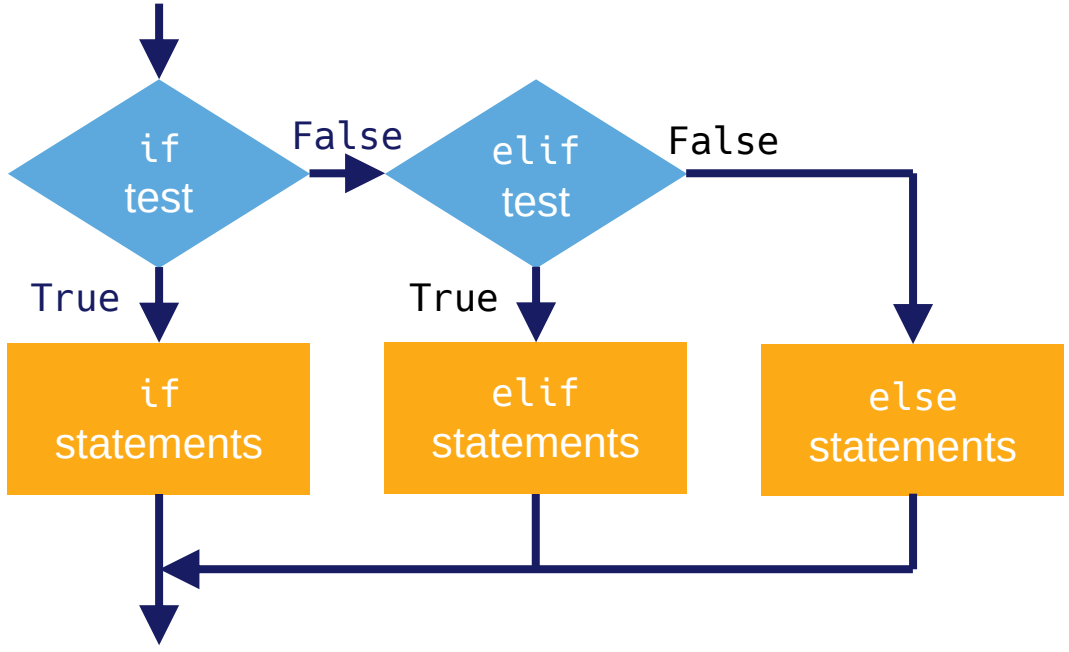
4.1. Conditional Expressions (3)

- `if`, `elif`, `else`

elif ⇒ else if

```
fruits = {'apple', 'orange', 'mango'}
vegetables = {'lettuce', 'cabbage', 'spinach'}
item = input('What are you offering? ')

if item in fruits:
    print('You are offering some fruits')
elif item in vegetables:
    print('You are offering some vegetables.')
else:
    print('I do not know what you are offering.')

print('Thanks for the offer!')
```

- Test the `if` expression, execute the indented `if` statements if `True`
- If `False`, test the following `elif` expression, execute the indented `elif` statements if `True`
- Execute the final `else` statements if none of the expressions are `True`



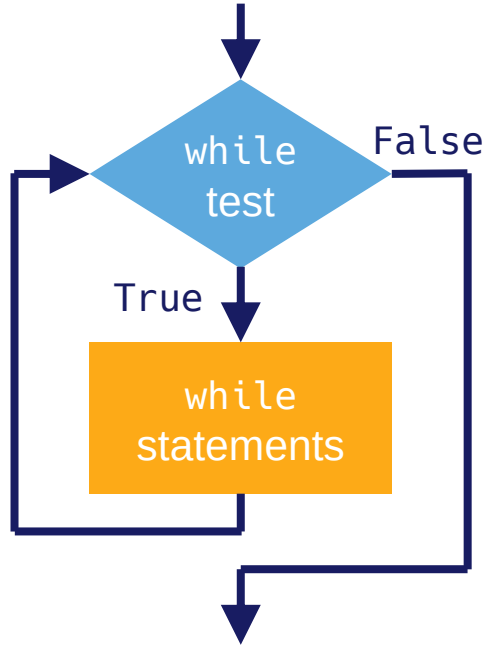
4.2. Indefinite Iteration

- Iteration – executing same block of code multiple times
- `while`

```
mark = 0

while mark<50:
    print('You have not passed the test yet.')
    mark = float(input('Enter the mark (0-100): '))

print('Congratulations! You have passed the test.')
```

- Test the `while` expression
- If `True`, execute the indented `while` statements, test the `while` expression again

- Repeat until the `while` expression evaluates to `False`, exit if `False`
- May result in an infinite loop, if the `while` expression never evaluates to `False` – Use `CTRL+C` to force exit



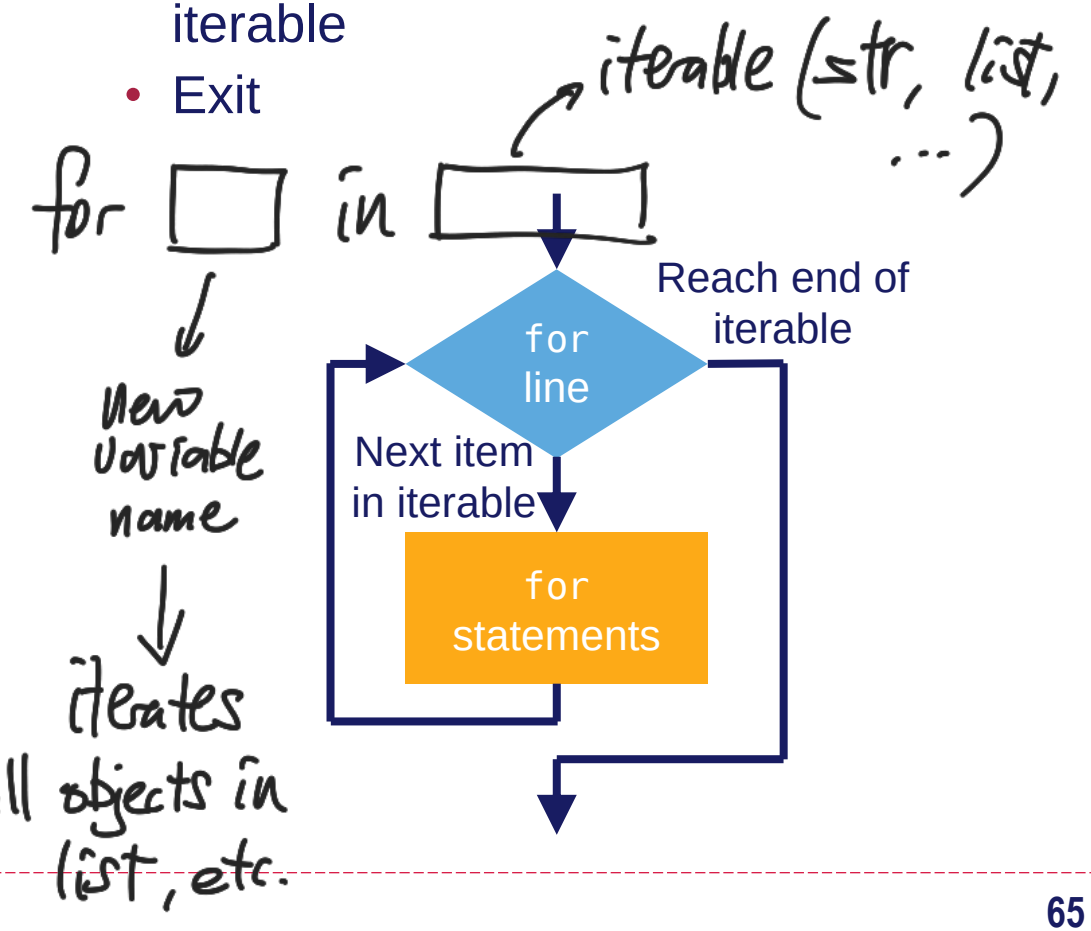
4.3. Definite Iteration

- Iterate over an iterable (`str`, `list`, `tuple`, `dict`, `set`): `for`

```
prime = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
sum = 0

for x in prime:
    print(x)
    sum += x

print('Sum of first', len(prime), 'primes:', sum)
```

  - Assign the first item in the iterable to the variable defined in the `for` line
  - Execute the indented `for` statements

- Repeat for the next item in the iterable until the indented `for` statements are executed for the last item in the iterable
- Exit



4.4. Nested Statements

- Achieve complex tasks by using nested conditional expressions and definite/indefinite iterations
- E.g. Print the first n prime numbers

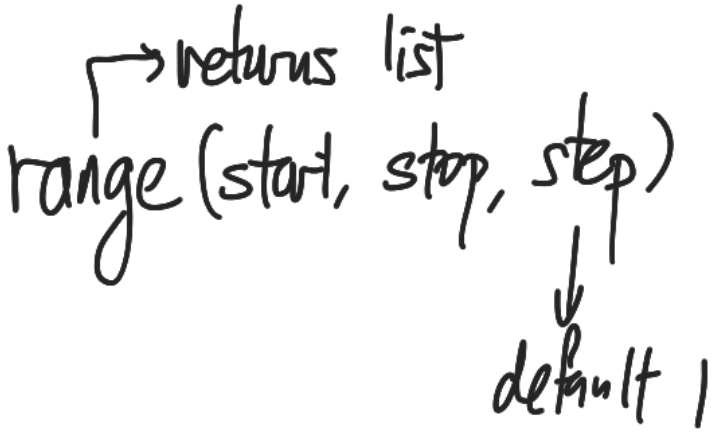
```
n = int(input('Number of primes to print: '))
prime_list = []
candidate = 2

while len(prime_list)<n:
    prime = True
    for x in range(2, candidate):
        if candidate%x==0:
            prime = False
    if prime:
        prime_list.append(candidate)
    candidate += 1

print(prime_list)
```

- Assign an empty `list`

4.4. Nested Statements

- Achieve complex tasks by using nested conditional expressions and definite/indefinite iterations
- E.g. Print the first n prime numbers

```
n = int(input('Number of primes to print: '))
prime_list = []
candidate = 2

while len(prime_list)<n:
    prime = True
    for x in range(2, candidate):
        if candidate%x==0:
            prime = False
    if prime:
        prime_list.append(candidate)
    candidate += 1

print(prime_list)
```



- `range(start, stop, step)` – A sequence of integer numbers from `start` at an increment of `step`, up to but NOT including `stop`

4.4. Nested Statements

- Achieve complex tasks by using nested conditional expressions and definite/indefinite iterations
- E.g. Print the first n prime numbers

```
n = int(input('Number of primes to print: '))
prime_list = []
candidate = 2

while len(prime_list)<n:
    prime = True
    for x in range(2, candidate):
        if candidate%x==0:
            prime = False
    if prime:
        prime_list.append(candidate)
    candidate += 1

print(prime_list)
```

- `int%int` return the remainder from the division

4.4. Nested Statements

- Achieve complex tasks by using nested conditional expressions and definite/indefinite iterations
- E.g. Print the first n prime numbers

```
n = int(input('Number of primes to print: '))
prime_list = []
candidate = 2

while len(prime_list)<n:
    prime = True
    for x in range(2, candidate):
        if candidate%x==0:
            prime = False
    if prime:
        prime_list.append(candidate)
    candidate += 1

print(prime_list)
```

- Continue `while` loop until `n` prime numbers are found
- Check if `candidate` is a prime number
- If `True`, append candidate to `prime_list`

5.
File Input and Output

5.1. Opening and Closing File

- Data is stored in files – File I/O (input/output) operations
- Open a file to perform an file I/O operations

```
>>> f = open('C:\\Users\\wonkeun.chang\\data.txt')
>>>
>>> imgfile = open('image.png', 'rb')
```

- Operating system dependent
- A backslash (\) is used in Python as an escape character in `str` – Need to use two backslashes (\\) to register a single backslash in `str`

- Use the full path if the file is NOT in the current working directory

Close file: f.close()

f = open('file path', 'mode')

↓ Python file object

↓ default: rt

5.1. Opening and Closing File

- Data is stored in files – File I/O (input/output) operations
- Open a file to perform an file I/O operations

! Return Python file object !

```
>>> f = open('C:\\Users\\wonkeun.chang\\data.txt')
>>>
>>> imgfile = open('image.png', 'rb')
```

  - Use the full path if the file is NOT in the current working directory
  - Mode (default: 'rt')

reading only, text file

- Close the file after completing operations

```
>>> f.close()
>>> imgfile.close()
```

'r'	Read (default)
'w'	Write: create a new file if the file does not exist, or overwrites the file if it exists
'x'	Write: create a new file if the file does not exist, or operation fails if it exists
'a'	Write: create a new file if the file does not exist, or append to the file if it exists
't'	Text (default)
'b'	Binary
'+'	Read and write ('r+', 'rb+', 'w+', 'wb+', 'a+', 'ab+')

5.2. Reading Text File (1)

- Example text file

Text example source: From *Sonnet 18* by W. Shakespeare

```
Shall I compare thee
to a summer's day?
```

- Read the entire file: `read`

```
>>> f = open('sonnet18.txt')
>>>
>>> whole = f.read()
>>> whole
"Shall I compare thee \nto a summer's day?"
>>>
>>> type(whole), len(whole), whole.index('\n')
(<class 'str'>, 40, 21)
>>>
>>> f.close()
```

- Create `sonnet18.txt` file in the Python's working directory

`str = f.read()` ⇒ returns string of entire file
↳ file object

- Read the entire file and assign it to `whole`
- Any white spaces at the end of lines are retained
- Newline character '`\n`' is a separate character in `str` (`whole[21]`)

5.2. Reading Text File (2)

- Read line by line: `readline`, `readlines`

Text example source: From *Sonnet 18* by W. Shakespeare

```
>>> f = open('sonnet18.txt')
>>>
>>> f.readline()
'Shall I compare thee \n'
>>> f.readline()
"to a summer's day?"
>>>
>>> f.seek(0)
0
>>> f.readlines()
['Shall I compare thee \n', "to a summer's day?"]
>>>
>>> f.close()
```

f.readline() ⇒

- Read a line up to and including the newline character (`'\n'`), repeat until EOF (end of file) is reached

f.seek(0) ⇒

- Move the current position (of pointer) of the file object to the beginning

f.readlines() ⇒

- Read the entire file into `list` (returns) of `str` with each line as its item

! `split` operation in `str` (Section 3.4.3) to separate text into words

Type casting (Section 3.5) to read numeric data from text file !

5.3. Writing to Text File (1)

- Write `str` to a file: `write`

```
verse1 = 'All that is gold does not glitter, \n'

f = open('tolkien.txt', 'w')

f.write(verse1)
f.write('Not all those who wander are lost.')

f.close()
```

```
All that is gold does not glitter,
Not all those who wander are lost.
```

Text example source: From *The Lord of the Rings* by J. R. R. Tolkien

'w'

- Overwrite if file exist – Use mode `'a'` to append to the existing file
- Write `str` to the file – `'\n'` send subsequent substring to the next line
- Next `write` call append `str` from the current position of the file object

f.write(str)

Content of `tolkien.txt` file created

5.3. Writing to Text File (2)

- E.g. Append rainfall record (24–30 June 2018) in Changi Airport

```
loc, year, month = 'Changi', 2018, 6

dates = [24, 25, 26, 27, 28, 29, 30]
data = [0, 15.2, 31.4, 10.6, 0, 6, 0]

fname = loc+str(year)+str(month).zfill(2)+'.txt'
f = open(fname, 'a')

for day, dailyfall in zip(dates, data):
    datestr = str(year)+str(month).zfill(2)+\
              str(day).zfill(2)
    datastr = '{:05.1f}'.format(dailyfall)
    f.write(datestr+' '+datastr+'\n')

f.close()
```

- Use `\` to continue a long expression in the next line

- Create a systematic file name for the data – `'Changi201806.txt'`
- Cast to `str`, fill it with zeros if its length is shorter than 2, e.g., '6' → '06'
  `str.zfill(_)`
- Append to the existing data file
  `zip(list1, list2)` ⇒ iterates 2 lists simultaneously
- Cast the numeric data to `str`
- Cast `float` to `str` of length 5 with a single decimal place, fill `str` with zeros if its length is shorter than 5

`'{:5.1f}'.format(data)`
→ float
returns string

Rainfall data source: http://www.weather.gov.sg/climate-historical-daily/

5.3. Writing to Text File (2)

- E.g. Append rainfall record (24–30 June 2018) in Changi Airport

```
loc, year, month = 'Changi', 2018, 6

dates = [24, 25, 26, 27, 28, 29, 30]
data = [0, 15.2, 31.4, 10.6, 0, 6, 0]

fname = loc+str(year)+str(month).zfill(2)+'.txt'
f = open(fname, 'a')

for day, dailyfall in zip(dates, data):
    datestr = str(year)+str(month).zfill(2)+\
              str(day).zfill(2)
    datastr = '{:05.1f}'.format(dailyfall)
    f.write(datestr+' '+datastr+'\n')

f.close()
```

Content appended to `Changi201806.txt`

```
20180624 000.0
20180625 015.2
20180626 031.4
20180627 010.6
20180628 000.0
20180629 006.0
20180630 000.0
```

Rainfall data source: http://www.weather.gov.sg/climate-historical-daily/

5.4. File I/O in Other Formats

- Storing large datasets to a text file is inefficient
- Some better alternatives
  - pickle – For serialising Python object directly in binary format
    - NOT for data that is to be used across different languages
    - Official documentation – https://docs.python.org/3/library/pickle.html
  - HDF5 – For handling very large numerical data independent of language
    - Slice into multi-TB data, no need to hold the entire data in the memory
    - Use a third-party package to interface with Python, e.g. *h5py* – http://docs.h5py.org/
  - JSON – Text-based language independent data-interchange format
    - Store standard Python objects directly in a human readable file
    - Official documentation – https://docs.python.org/3/library/json.html

6.
Errors and Exceptions

6.1. Syntax Error and Exception Error (1)

- Syntax error – Grammatical error

```
>>> for i in range(3)
  File "<stdin>", line 1
    for i in range(3)
                     ^
SyntaxError: invalid syntax
```

  - Missing a colon (`:`) syntactically required at the end of the `for` loop line
  - Cannot be “handled” – Must be corrected in the code

6.1. Syntax Error and Exception Error (2)

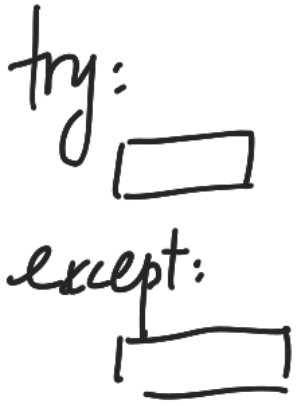
- Exception error – Run-time error

```
>>> '5'+4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

- No issues grammatically, but encountered error during the execution
- Incorrect Data Type detected – Expected `str` flowing `'5'` for `str` concatenation
- Can be "handled" on run-time

6.2. Exception Handling (1)

- Handle any exception errors: `try, except`



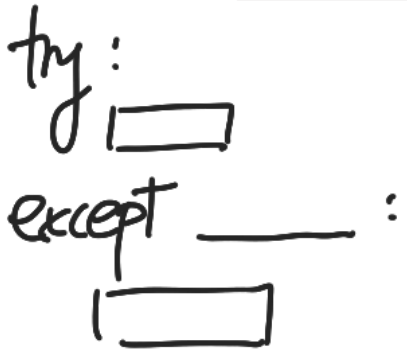

```
a, b = '5', 4

try:
    print(a+b)
except:
    print('An error occurred.')
```

  - Execute the indented `try` statements
  - If ANY exception error occur in `try` statements, leave the block and execute the indented `except` statements

```
An error occurred.
```

6.2. Exception Handling (2)

- Handle specific exception errors: `try`, `except ExceptionName`




```
a, b = '5', 4

try:
    print(a+b)
except TypeError:
    print(int(a)+int(b))
```

  - Execute the indented `try` statements
  - If `ExceptionName` error occur in `try` statements, leave the block and execute the indented `except` statements

```
9
```

6.2. Exception Handling (3)

- Handle exception errors differently

```
try:
    int('string')
except TypeError:
    print('A TypeError occurred.')
except ValueError:
    print('A ValueError occurred.')
except:
    print('An error occurred')
```

Can stack together (like if)

- Execute specific `except` statements depending on the type of error encountered in `try` statements

```
A ValueError occurred.
```

6.2. Exception Handling (4)

- Some common `ExceptionName`

`KeyError`	Key is not found in a dictionary
`IndexError`	Index not found in the sequence
`IOError`	I/O operation failed
`NameError`	Identifier not found in the namespace
`TypeError`	Operation invalid for Data Type
`ValueError`	Invalid value for specified operation
`ZeroDivisionError`	Division by zero for all numeric types

6.2. Exception Handling (5)

- `try`, `except`, `else`

```
string = '3'

try:
    number = int(string)
except:
    print('An error occurred.')
else:
    print('It was a success!')
```

  - Execute the indented `else` statements ONLY if no exceptions are encountered

```
It was a success!
```

6.2. Exception Handling (6)

- `try`, (`except`), (`else`), `finally`

```
try:
    number = int('3.0')
except:
    print('An error occurred.')
else:
    print('It was a success!')
finally:
    print('I hope you enjoyed my code.')
```

  - Always execute the indented `finally` statements regardless of whether an exception was encountered or not

int('3.0') → error
∵ have decimal, only can convert to float

```
An error occurred.
I hope you enjoyed my code.
```

6.3. Exception Raising (1)

- Manually raise an exception: `raise`

```
x = -1

if x<0:
    raise ValueError('x must be positive.')
```

  - E.g. If `x` must be a positive value, `ValueError` exception can be manually raised with a custom error message

```
Traceback(most recent call last):
  File "raise.py",  line 4, in <module>
    raise ValueError('x must be positive.')
ValueError: x must be positive.
```

6.3. Exception Raising (2)

- Putting it all together…

```
numberstr = input('Enter a number between 0 and 100: ')

try:
    number = float(numberstr)
    if number<0:
        raise RuntimeError
    elif number>100:
        raise RuntimeError
except RuntimeError:
    print('Your number is outside of the range.')
except ValueError:
    print('That is not a number.')
else:
    print(numberstr+' is successfully registered.')
```

- Type casting from `str` to `float` raise `ValueError` exception if `numberstr` is non-numeric
- Manually raise `RuntimeError` exception if the input is outside of the specified range
- If no exceptions are raised, notify that the input is successfully registered in `number` as `float` Data Type within the specified range

References

- Official Python documentation

  https://docs.python.org/3/

- Official Python tutorial

  https://www.python.org/3/tutorial

- Official Python standard library reference

  https://www.python.org/3/library

- Official Anaconda Distribution documentation

  https://docs.anaconda.com/anaconda/