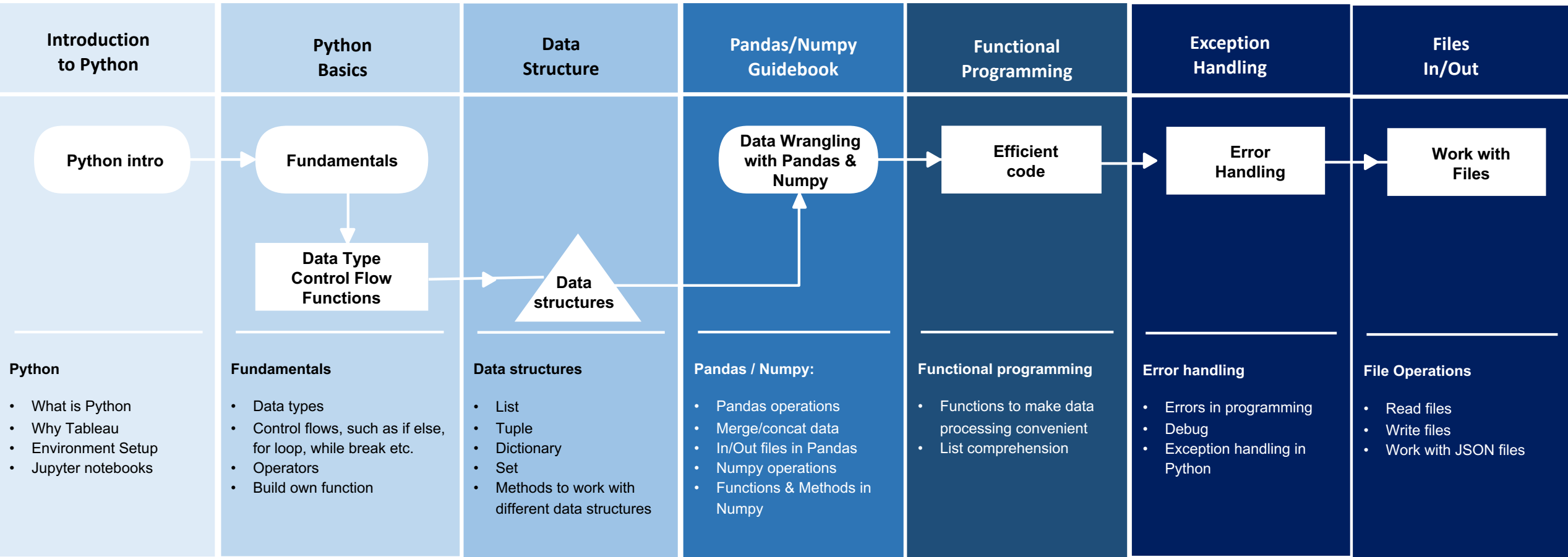




Data Analysis with Python

COURSE AGENDA



Goals

- ✓ Master Python basics to build efficient programs
- ✓ Learn to use Pandas and Numpy to analyze data sets
- ✓ Learn to process files and handle errors to automate daily routine works

1.1 Intro to Python

What is python?

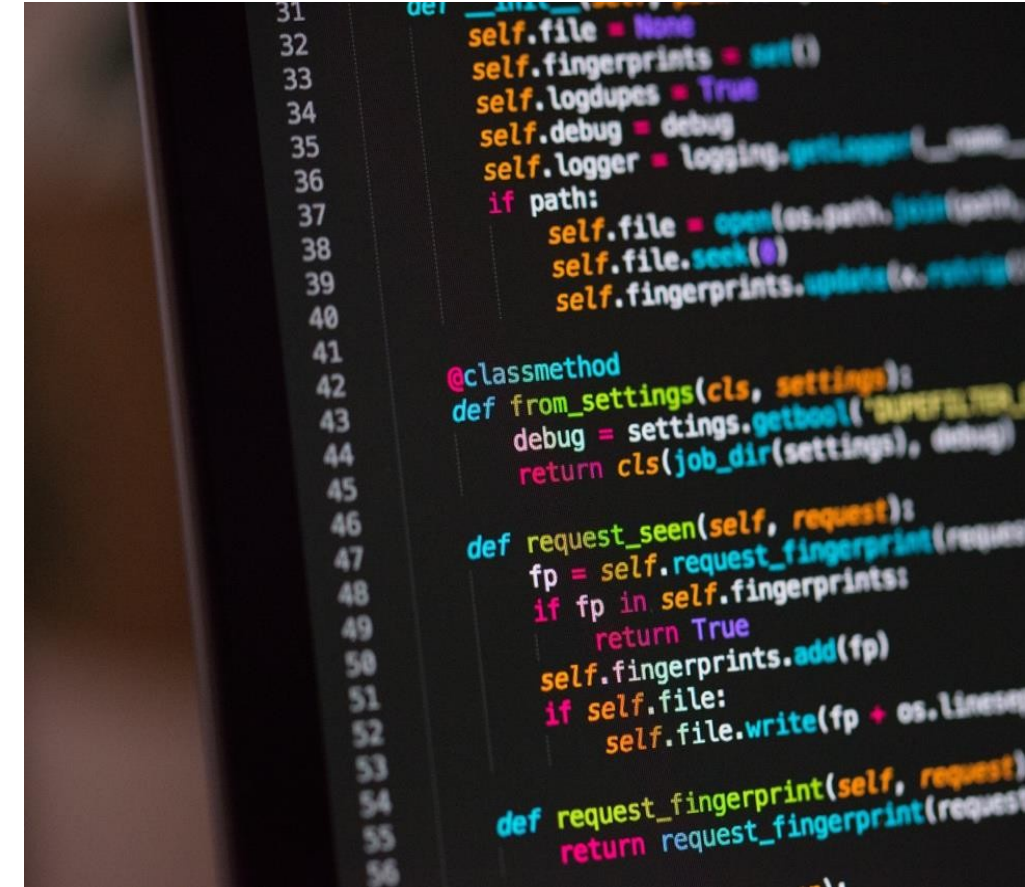
Python is an interpreted, high-level and general-purpose programming language. Python's design philosophy emphasizes code readability with its notable use of significant indentation.

What can python do?

- Software and web application development
- Data processing, scientific computing
- Machine learning, artificial intelligence
- System scripting, robotic processing
- ...

Why python?

Python is easy to use, powerful, and versatile, making it a great choice for beginners and experts alike. Python's readability makes it a great first programming language.

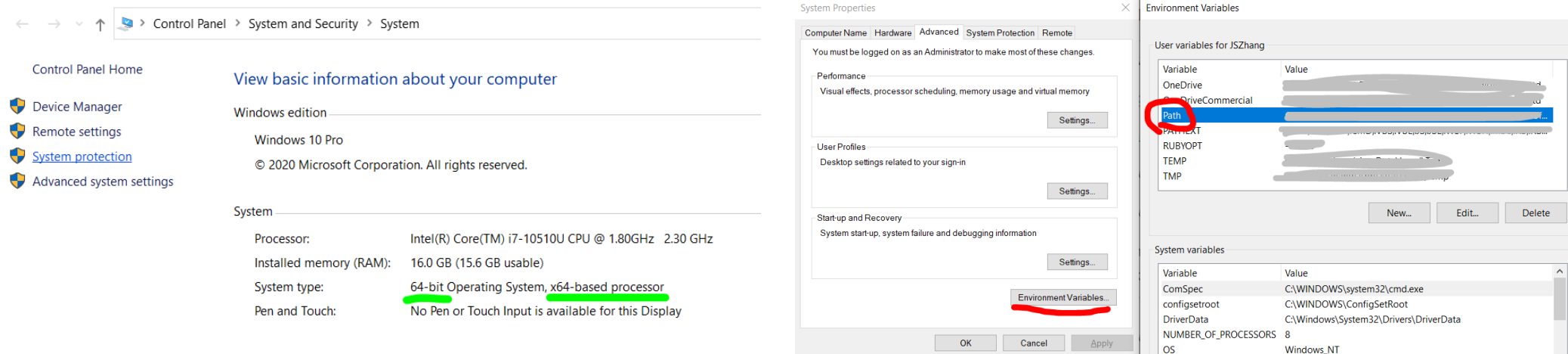


```
31
32
33     self.file = None
34     self.fingerprints = self()
35     self.logdups = True
36     self.debug = debug
37     self.logger = logging.getLogger(__name__)
38     if path:
39         self.file = open(os.path.join(path,
40                                     self.file.seek(0)
41                                     self.fingerprints.update(s.request)
42
43 @classmethod
44 def from_settings(cls, settings):
45     debug = settings.getbool('debug')
46     return cls(job_dir(settings), debug)
47
48 def request_seen(self, request):
49     fp = self.request_fingerprint(request)
50     if fp in self.fingerprints:
51         return True
52     self.fingerprints.add(fp)
53     if self.file:
54         self.file.write(fp + os.linesep)
55
56 def request_fingerprint(self, request):
57     return request_fingerprint(request)
```

1.2 Install Python on Windows

Windows

Download Python installer from website (<https://www.python.org/downloads/windows/>) based on your Windows version (32-bit or 64-bit). Run the downloaded installer program, remember to select Add Python 3.x to PATH before clicking Install Now



To run Python3 REPL (interactive program, REPL stands for Read-Evaluate-Print-Loop), type python in Command Prompt.

```
C:\Users\jszhang>python
Python 3.8.8 (tags/v3.8.8:024d805, Feb 19 2021, 13:18:16) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Windows

If you see error message 'python' is not recognized as an internal or external command, then add the path to python.exe to environment variables

1.2 Install Python on MacOS

Method 1:

Download Python installer from website (<https://www.python.org/downloads/mac-osx/>).

Method 2:

Run command **brew install python3** if *Homebrew* is installed

To run Python3 REPL, type python3 in Terminal application.

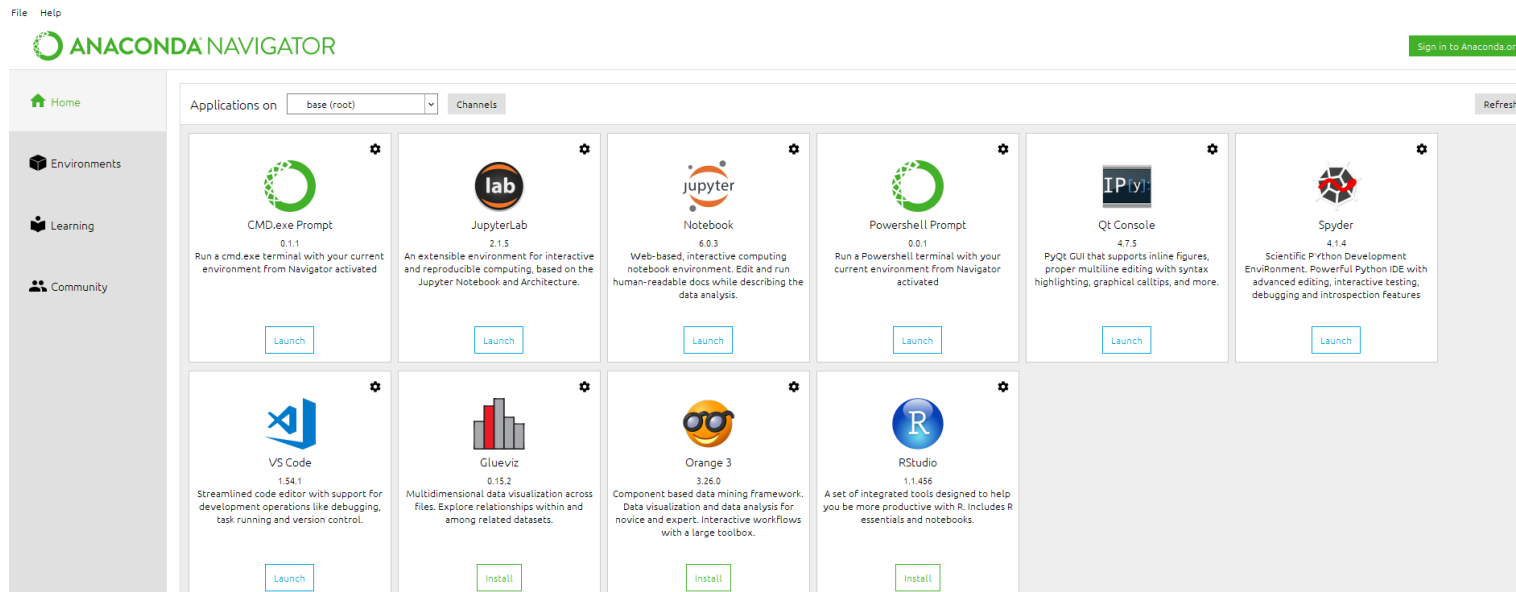
```
→ ~ python3
Python 3.9.0 (v3.9.0:9cf6752276, Oct 5 2020, 11:29:23)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

1.3 Install Jupyter Notebook via Anaconda

Anaconda

Anaconda is a free and open-source distribution of the Python programming languages for scientific computing that aims to simplify package management and deployment. Visit <http://anaconda.com/downloads> to download and install Anaconda

- It provides different interface (Jupyter, Spyder) to execute python functions.
- Python automatically comes with Anaconda; hence users only need to install Anaconda.
- Python and pip (Python package manager) are included in Anaconda. After installation, the default python path is modified by Anaconda



Jupyter Notebook App

It is a server-client application that allows editing and running notebook documents via a web browser.

Spyder

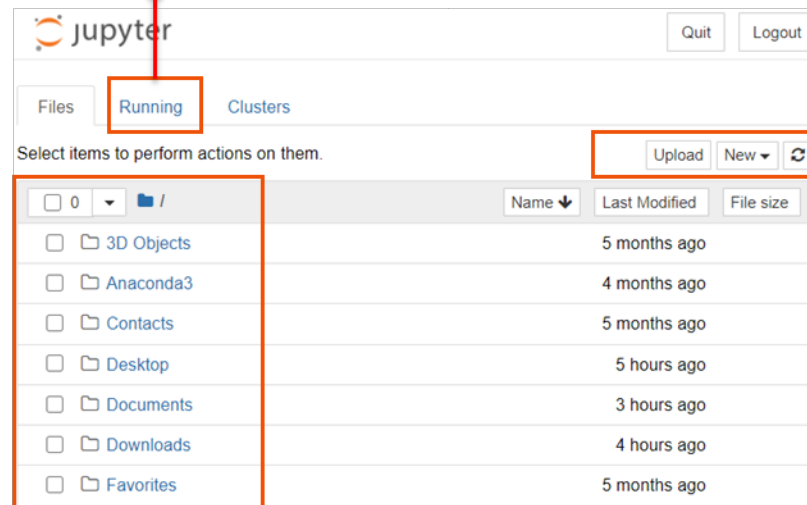
Spyder is an open-source cross-platform integrated development environment (IDE) for scientific programming in the Python language.

1.3 Jupyter Notebook Interface

- Jupyter notebook opens to the default file directory (C: drive).
- Open existing files by navigating to the appropriate directory or create a new python file using the “New” button in the top right corner.

Running

Displays the current open python files running in Jupyter notebook

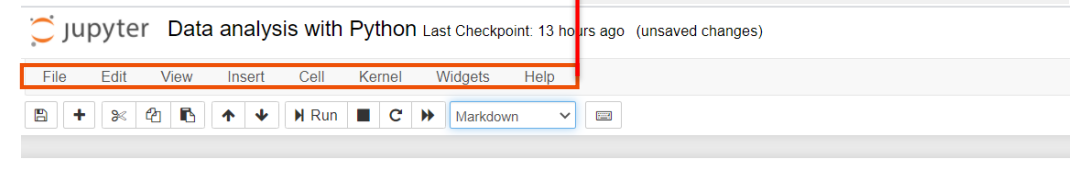


New

To create a new python notebook files, folder etc.

Toolbar

Allows users to run, add another section of code, or change the code format.



Data Analysis with Python

ver 1.0 (Mar 2021)
by Jason Zhang

1 Introduction

- [1.1 What is Python](#)
- [1.2 Install Python3](#)
- [1.3 Jupyter Notebook](#)

Input/Output

User input of code or markdown and the result of output

File Directory

Navigation of existing python notebook files to open

1.3 Jupyter Notebook Interface

Jupyter Notebook Shortcuts

Edit mode: text area is focused; left border is **green**.

Command mode: text area is out of focus, left border is **blue**.

Press Esc to exit Edit mode and enter Command mode . Use up/down arrow keys to select different lines.

Tips:

to clean a cell's output,

select the target cell, then press `Esc` , `r` , `y` in sequence.

Command Mode (press `Esc` to enable)

`F`: find and replace

`Ctrl-Shift-F`: open the command palette

`Ctrl-Shift-P`: open the command palette

`Enter`: enter edit mode

`P`: open the command palette

`Shift-Enter`: run cell, select below

`Ctrl-Enter`: run selected cells

`Alt-Enter`: run cell and insert below

`Y`: change cell to code

`M`: change cell to markdown

`R`: change cell to raw

`1`: change cell to heading 1

`2`: change cell to heading 2

`3`: change cell to heading 3

`4`: change cell to heading 4

`5`: change cell to heading 5

`6`: change cell to heading 6

`K`: select cell above

`Up`: select cell above

`Down`: select cell below

`J`: select cell below

`Shift-K`: extend selected cells above

`Shift-Up`: extend selected cells above

`Shift-Down`: extend selected cells below

`Shift-J`: extend selected cells below

`A`: insert cell above

`B`: insert cell below

`X`: cut selected cells

`C`: copy selected cells

`Shift-V`: paste cells above

`V`: paste cells below

`Z`: undo cell deletion

`D`, `D`: delete selected cells

`Shift-M`: merge selected cells, or current cell with cell below if only one cell is selected

`Ctrl-S`: Save and Checkpoint

`S`: Save and Checkpoint

`L`: toggle line numbers

`O`: toggle output of selected cells

`Shift-O`: toggle output scrolling of selected cells

`H`: show keyboard shortcuts

`I`, `I`: interrupt the kernel

`0`, `0`: restart the kernel (with dialog)

`Esc`: close the pager

`Q`: close the pager

`Shift-L`: toggles line numbers in all cells, and persist the setting

`Shift-Space`: scroll notebook up

`Space`: scroll notebook down

2.1 Print

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

```
Hello World!
```

In Python3, `print` is a function.

A function can be invoked by specifying the function name, followed by arguments enclosed by a pair of parentheses.

We will look at more about functions later.

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

If no argument is given, `print` will output an empty line.

```
In [3]: print("1 + 2 =", 1 + 2)
```

```
1 + 2 = 3
```

The print function outputs 2 parameters:

- string "1 + 2 ="
- the evaluated value of expression 1 + 2, which is 3

2.2 Variables

Variable is also called **Identifier**, which is a name referring to a value or values stored in the memory of the computer. We can imagine there is a look-up table stored inside the computer, every time we define a variable, Python inserts one key-value pair into that table, where key is our variable name, and value is simply the value that the variable name referring to.

```
In [1]: name = "Jason Zhang"
        print(name)
```

Jason Zhang

```
In [2]: a = 'abc'
        a = 3
        print(a)
```

3

```
In [3]: a = 3
        b = a = 5
        print(a, b)
```

5 5

```
In [4]: a, b, c = 12, "Wow!", False
        print(a, b, c)
```

12 Wow! False

- We created a variable called name and assigned the value "Jason Zhang" to it.
- The first line of the program is called an **assignment statement**.
name ← "Jason Zhang"
- Variables' values can be changed (re-assigned).
- Variable a is defined with an initial value 'abc'. Then a new value 3 is assigned to the variable a, thus the value of a is 3 at the final state.
- Assignment statement is evaluated from right to left.
- The expression on the right-hand-side of = is evaluated first, then assigned to the variable name on the left-hand-side of =.
- You can assign values to multiple variables within one assignment statement

| 2.2 Variables – code challenge

Q: Write a program to swap two variables.

2.3 Comments

Annotates the code to help the programmer understands the intended logic.

These comments are ignored by the interpreter. A comment line starts with #.

Try to add some meaningful comments for reference, not only for you, but also someone who may read your code.

```
salary = salary * 1.02  # increase salary by 2%
```

```
def increase(salary, percentage, rating):  
    """ increase salary base on rating and percentage  
    rating 1 - 2 no increase  
    rating 3 - 4 increase 5%  
    rating 4 - 6 increase 10%  
    """
```

2.4 Data Types

Data Types	Description	Example
String (str)	A string form, usually printed with inverted commas, applicable to all words including numbers	"a", "good", "TRUE", "23.4", "2"
Integer (int)	A number that is usable with all aggregate and non- aggregated math functions	1, 2, 3
Float (float)	Like numeric, except it has decimal place, i.e., not a whole number	1.23, 3.5
Logical (bool)	A Boolean expression, defining true or false	True, False (Note: this is case-sensitive, hence it must be typed exactly as this example)
None (None)	None stands for an empty value	None

Use **type()** to get the data type of a specific value or a variable.

```
In [5]: type_1 = type(123)
type_2 = type(123.0)
type_3 = type("123")
type_4 = type(True)
type_5 = type(print)
type_6 = type(None)

print("type_1 =", type_1)
print("type_2 =", type_2)
print("type_3 =", type_3)
print("type_4 =", type_4)
print("type_5 =", type_5)
print("type_6 =", type_6)
```

```
type_1 = <class 'int'>
type_2 = <class 'float'>
type_3 = <class 'str'>
type_4 = <class 'bool'>
type_5 = <class 'builtin_function_or_method'>
type_6 = <class 'NoneType'>
```

2.4 Data Types

Integers (int)

Python can deal with any integer values, including negative integers, such as 0, -99, 12345

Floating Point Number (float)

A floating-point number is a number that has a fractional part.

It is called floating point number because the position of the decimal point can be changed in the scientific notation.

For example, 3.1415e9 is the same as 314.15e7.

```
In [6]: print(5.1 + 2.3)
```

```
7.3999999999999995
```

```
In [7]: round(5.1 + 2.3, 2) # round to 2 digits precision after the decimal point
```

```
Out[7]: 7.4
```

String (str)

A string is a text surrounded by a pair of single quotes ' or double quotes ", for example 'This is a string.', "This is another string.".

The first and last quotation marks should be matched, and they are not parts of the string.

Use escape character (\) to output new line \n, tab \t, back slash \, single quote \', double quote \" etc. inside a string.

```
In [8]: print('First line\nSecond line\nI\'m the third line')
```

```
First line
Second line
I'm the third line
```

```
In [9]: print('This is \\ a string.')
print(r'This is \\ a string.')
```

```
This is \ a string.
This is \\ a string.
```

2.5 String Formatting

During programming, we are required to form strings containing variables frequently. We can do this with concatenation using `+` operator.

```
In [1]: name = 'Tom'
        age = 23

        string = name + ' is ' + str(age) + ' years old'
        print(string)

Tom is 23 years old
```

Note:

1. A string cannot concatenate with a non-string data type; thus we must cast non-string values using **str** function.
2. We need to add leading or trailing spaces to some strings in order to make sure words are separated by spaces properly.

In Python, each string has a format function. **format_string.format(arguments)**.

```
In [2]: string = '{} is {} years old'.format(name, age)
        print(string)

Tom is 23 years old
```

Note:

1. `{}` represents a variable in the format string.
2. We can re-arrange the order by specifying the index number in the braces, for example `{2}`.

We can apply padding, aligning and truncating to formatting numbers.

```
In [3]: '{:06.2f}'.format(3.14159265358)

Out[3]: '003.14'
```

Note:

1. Use letter **f** represent floating point numbers.
2. We add a number after `.` to limit the number of digits after the decimal point

2.6 Operators

Basic Operators

Operator	Description	Example
+	Addition	<code>3 + 5 ⇒ 8</code>
-	Subtraction	<code>5 - 3 ⇒ 2</code>
*	Multiplication	<code>5 * 3 ⇒ 15</code>
/	Division	<code>9 / 2 ⇒ 4.5</code>
//	Floor Division	<code>9 // 2 ⇒ 4</code>
%	Modulus	<code>9 % 2 ⇒ 1</code>
**	Exponent	<code>2 ** 3 ⇒ 8</code>

Assignment Operators

Operator	Example
=	<code>a = 10</code>
+=	<code>a += 5 ⇒ a = a + 5</code>
-=	<code>a -= 5 ⇒ a = a - 5</code>
*=	<code>a *= 5 ⇒ a = a * 5</code>
/=	<code>a /= 5 ⇒ a = a / 5</code>
//=	<code>a //= 5 ⇒ a = a // 5</code>
%=	<code>a %= 5 ⇒ a = a % 5</code>
**=	<code>a **= 5 ⇒ a = a ** 5</code>

Logical Operators

Operator	Description
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal
==	equal
!=	not equal
in	is in sequence
not in	is not in sequence

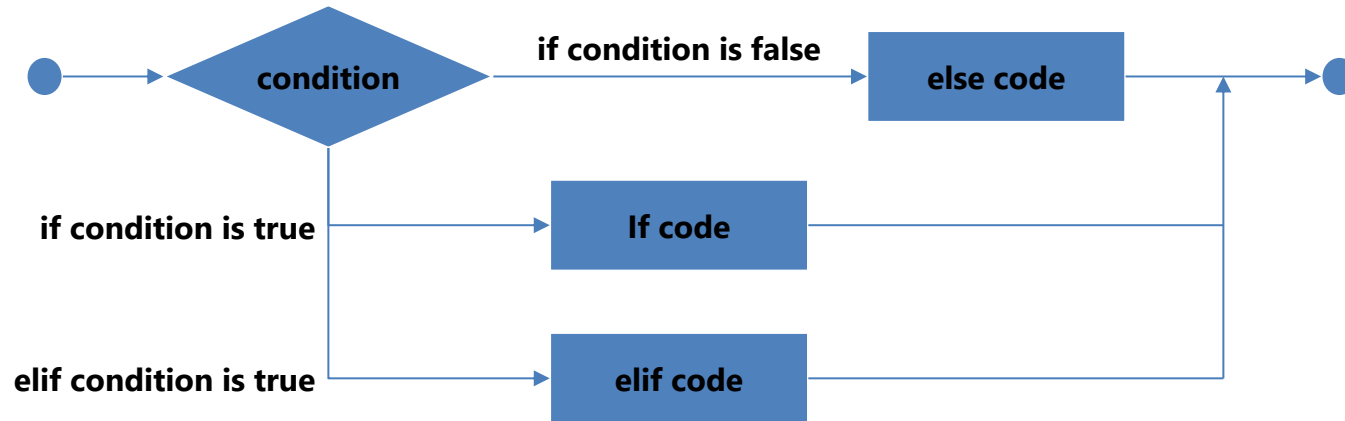
* A sequence can be a list, tuple etc.

Boolean Operators

Operator	Description	Example
and	Both conditions are True ⇒ result is True	<code>True and True ⇒ True</code>
or	At least one of the conditions is true ⇒ result is True	<code>False or True ⇒ True</code>
not	Negate the boolean value	<code>not True ⇒ False</code>

2.7 Control flow – if statement

if/elif/else Statement



```
In [4]: score = 70
if score >= 60:
    print('Pass')
```

Pass

```
In [5]: score = 50
if score >= 60:
    print('Pass')
else:
    print('Fail')
```

Fail

```
In [6]: score = 75
if score >= 90:
    print('A')
elif score >= 80:
    print('B')
elif score >= 70:
    print('C')
elif score >= 60:
    print('D')
else:
    print('F')
```

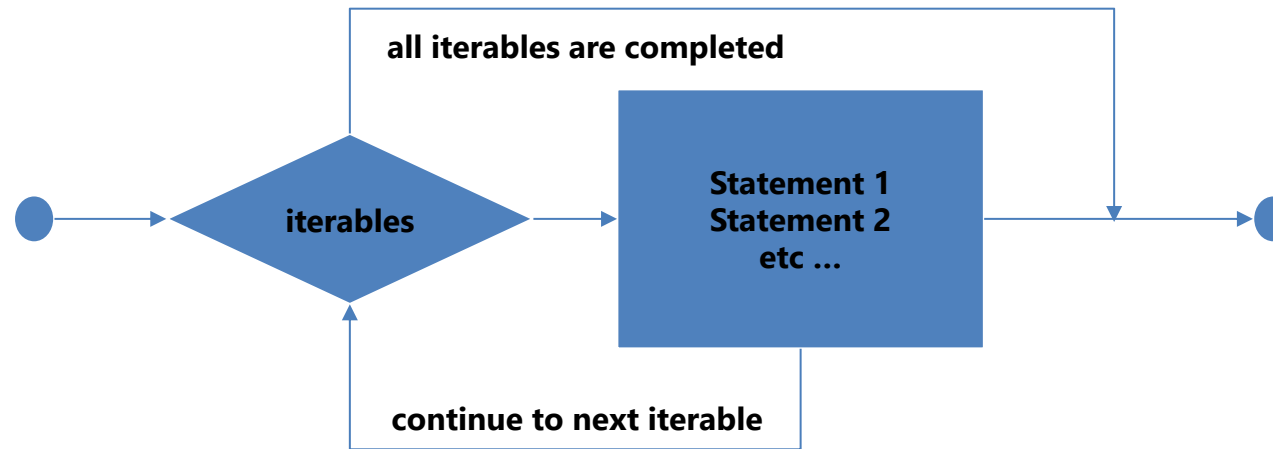
C

Note:

- Be careful to the indentation. Usually, we use 4-space or Tab indentation.
- Don't miss the colon :

2.7 Control flow – for loop

for loop



```
In [7]: countries = ['Singapore', 'Vietnam', 'Thailand', 'Indonesia']  
for country in countries:  
    print(country)
```

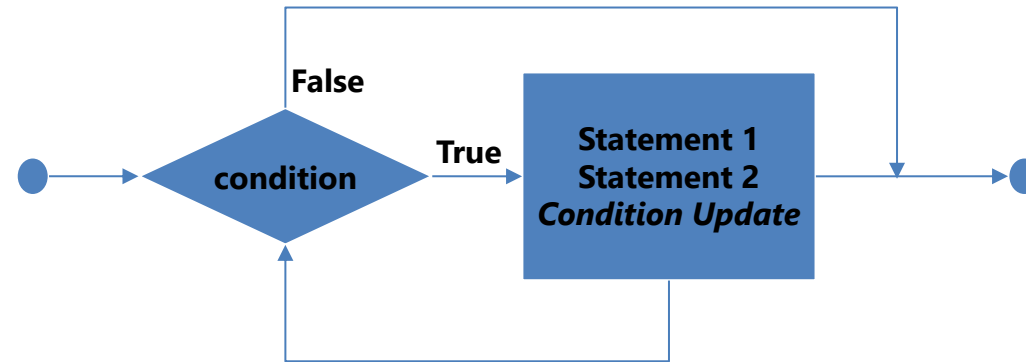
```
Singapore  
Vietnam  
Thailand  
Indonesia
```

```
In [8]: sum = 0  
for x in range(101):  
    sum += x  
    print(sum)
```

```
5050
```

2.7 Control flow – while loop

while loop



```
In [9]: def factorial(n):  
        fact = n  
        while n > 1:  
            n -= 1  
            fact *= n  
        return fact  
  
print(factorial(5))
```

120

```
In [10]: i = 0  
while i < 5:  
    print("i is {}".format(i))  
    i = i + 1
```

```
i is 0  
i is 1  
i is 2  
i is 3  
i is 4
```

Note:

- Be careful of infinite loops.

| 2.7 Control flow – code challenge

Q: Write a program to get sum of all the odd numbers within 1000 using loops in three ways.

2.7 Control flow – break/continue

break

use break statement to exit a loop.

```
In [11]: for x in [2, 4, 6, -1, 8, 0]:  
        if x < 0:  
            break  
        print(x)  
        print('END')
```

```
2  
4  
6  
END
```

continue

use continue to skip current iteration and continue with the next one.

```
In [12]: sum = 0  
        for x in [2, -2, 4, 6, -1, 8, 0]:  
            if x < 0:  
                print('skip ', x)  
                continue  
            sum += x  
        print('END')  
        print('result =', sum)
```

```
skip -2  
skip -1  
END  
result = 20
```

2.8 Built-in Functions

Some commonly used built-in functions

int: Return an integer object constructed from a number or string x, or return 0 if no arguments are given.

float: Return a floating point number constructed from a number or string x.

str: Return a string version of object.

bool: Return a Boolean value, i.e. one of True or False.

isinstance: Return true if the object argument is an instance of the classinfo argument, or of a (direct, indirect or virtual) subclass thereof.

abs: Return the absolute value of a number.

max: Return the largest item in an iterable or the largest of two or more arguments.

min: Return the smallest item in an iterable or the smallest of two or more arguments.

len: Return the length (the number of items) of an object.

range: The range type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in for loops.

type: With one argument, return the type of an object.

round: Return number rounded to ndigits precision after the decimal point.

map: Return an iterator that applies function to every item of iterable, yielding the results.

filter: Construct an iterator from those elements of iterable for which function returns true.

Type conversion

```
In [1]: float(7)
```

```
Out[1]: 7.0
```

```
In [2]: int(3.14)
```

```
Out[2]: 3
```

```
In [3]: str(3.14)
```

```
Out[3]: '3.14'
```

```
In [4]: float('3.14')
```

```
Out[4]: 3.14
```

```
In [5]: bool('3.14')
```

```
Out[5]: True
```

2.9 Functions

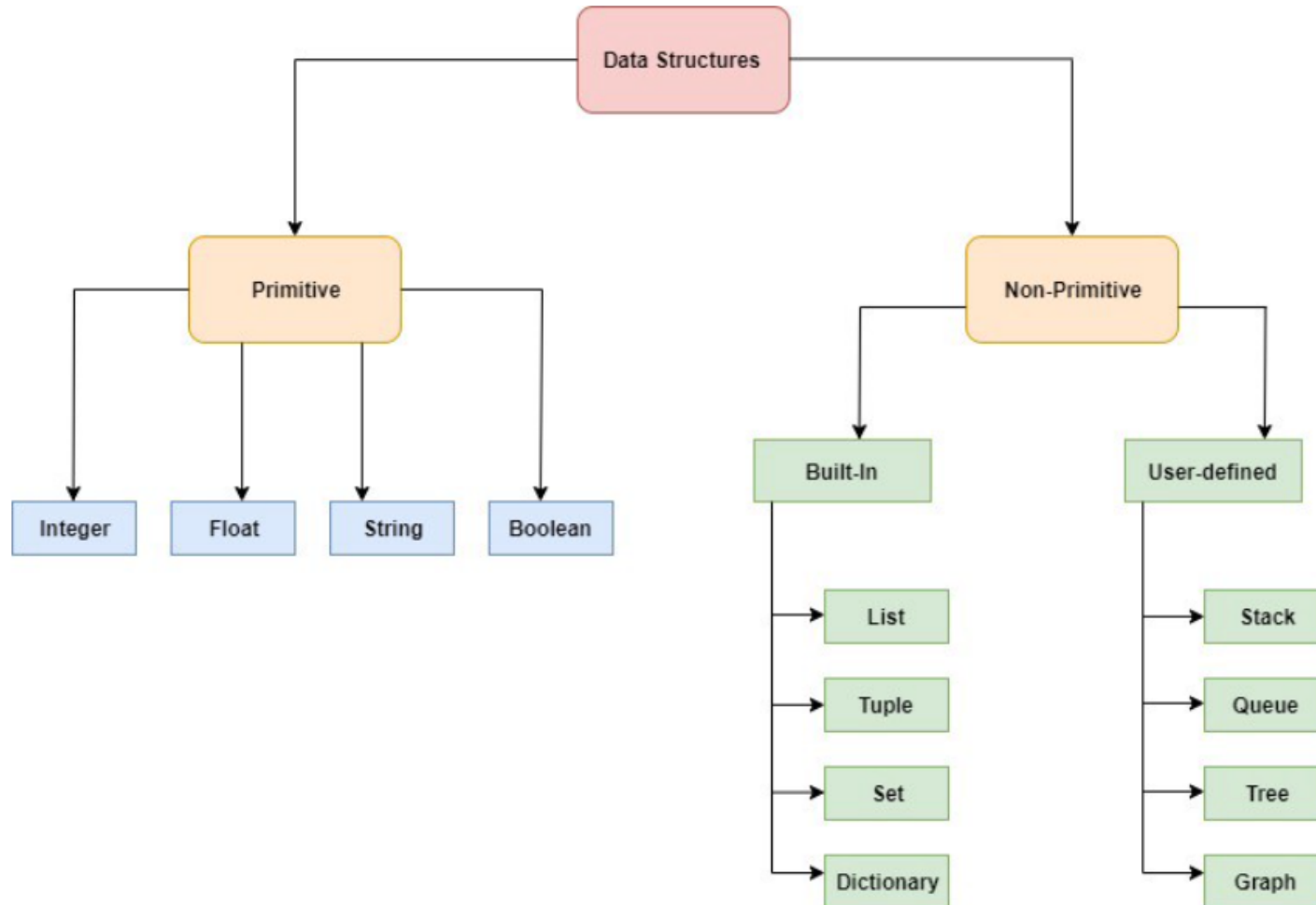
- Functions are reusable pieces of programs.
- A function can take parameters, which are values you supply to the function so that the function can *do* something utilising those values.

```
In [6]: def say_hello():  
        # block belonging to the function  
        print('hello world')  
        # End of function  
  
say_hello() # call the function  
say_hello() # call the function again  
  
hello world  
hello world
```

```
In [8]: def print_max(a, b):  
        if a > b:  
            print(a, 'is maximum')  
        elif a == b:  
            print(a, 'is equal to', b)  
        else:  
            print(b, 'is maximum')  
  
        # directly pass literal values  
print_max(3, 4)  
  
x = 5  
y = 7  
  
        # pass variables as arguments  
print_max(x, y)  
  
4 is maximum  
7 is maximum
```

3.1 Data structure

Data structures provide us with a specific and way of storing and organizing data such that they can be easily accessed and worked with efficiently.



3.2 List

- A list is a sequential collection of Python data values, where each value is identified by an index.
- The values that make up a list are called its elements.
- Lists are like strings, which are ordered collections of characters, except that the elements of a list can have any type and for any one list, the items can be of different types.

```
In [7]: vocabulary = ["iteration", "selection", "control"]
        numbers = [17, 123]
        empty = []
        mixedlist = ["hello", 2.0, 5*2, [10, 20]]

        print(numbers)
        print(mixedlist)
        newlist = [ numbers, vocabulary ]
        print(newlist)

[17, 123]
['hello', 2.0, 10, [10, 20]]
[[17, 123], ['iteration', 'selection', 'control']]
```

```
In [8]: len(vocabulary)
```

```
Out[8]: 3
```

The variable `vocabulary` is a list.
You can get the number of items in it by using **`len()`**

3.3 Access list element

- A list index starts from zero.
- We can get the index number of the last element in a list by calculating `len(list) - 1`. or
- we can use `-1` to get the last index directly

```
In [9]: numbers = [17, 123, 87, 34, 66, 8398, 44]
print(numbers[0])
print(numbers[9 - 8])
print(numbers[-2])
print(numbers[len(numbers) - 1])
print(numbers[7]) # out of range, will cause error
```

```
17
123
8398
44
```

```
-----
IndexError                                Traceback (most recent call last)
<ipython-input-9-dc4b21ffdba2> in <module>
      4 print(numbers[-2])
      5 print(numbers[len(numbers) - 1])
----> 6 print(numbers[7]) # out of range, will cause error

IndexError: list index out of range
```

```
In [10]: print(numbers[-1])
```

```
44
```

3.4 List slicing

It is a common operation to get a part of a list. For instance, to get the first 3 items in the following list.

```
In [4]: fruit = ['apple', 'banana', 'organge', 'cherry', 'kiwi']
```

```
In [5]: fruit[0:3]
```

```
Out[5]: ['apple', 'banana', 'organge']
```

```
In [6]: # from the beginning to the 3rd item -> the first 3 items  
fruit[:3]
```

```
Out[6]: ['apple', 'banana', 'organge']
```

```
In [7]: # from the second last to the end -> the last 2 items  
fruit[-2:]
```

```
Out[7]: ['cherry', 'kiwi']
```

```
In [9]: # starting at 0 and continuing until end,  
# take every other item from `fruit`  
fruit[::2]
```

```
Out[9]: ['apple', 'organge', 'kiwi']
```

Note:

1. `fruit[0:3]` means getting the items from index 0 to index 3, but **3 is not included**. Thus, the result is equivalent to `[fruit[0], fruit[1], fruit[2]]`

Note:

1. We can omit the starting index if it is 0.

Note:

1. Since `fruit[-2]` represent the second last item, we can apply the same idea in slicing.

Note:

1. If the ending index is not included, slicing will get until the end of the list. Remember that -1 index represents the last item.
2. If there is a number after two colons :, it represents steps.

3.5 List method

list is a mutable sequence, which means you can add items to it, or remove items from it

append(): To add one more item to the end of the list

```
In [1]: fruit = ['apple', 'banana', 'orange', 'cherry', 'kiwi']
fruit.append('coconat')
print(fruit)

['apple', 'banana', 'orange', 'cherry', 'kiwi', 'coconat']
```

insert(): Insert an item to a specific position in the list

```
In [4]: fruit = ['apple', 'banana', 'oranges', 'cherry', 'kiwi']
fruit.insert(2, 'mango')
print(fruit)

['apple', 'banana', 'mango', 'oranges', 'cherry', 'kiwi']
```

extend()

```
In [2]: fruit.extend(['avocado', 'mango'])
print(fruit)

['apple', 'banana', 'orange', 'cherry', 'kiwi', 'coconat', 'avocado', 'mango']
```

```
In [3]: fruit = ['apple', 'banana', 'orange', 'cherry', 'kiwi', 'coconat'] + ['avocado', 'mango']
print(fruit)

['apple', 'banana', 'orange', 'cherry', 'kiwi', 'coconat', 'avocado', 'mango']
```

pop(): Remove the last item

```
In [5]: fruit = ['apple', 'banana', 'orange', 'cherry', 'kiwi']
print(fruit.pop()) # pop() returns the popped item
print(fruit)

kiwi
['apple', 'banana', 'orange', 'cherry']
```

3.6 List & loops

Traverse the list using iteration by item

```
In [10]: fruits = ["apple", "orange", "banana", "cherry"]  
  
for afruit in fruits:    # by item  
    print(afruit)  
  
apple  
orange  
banana  
cherry
```

Traverse the list using iteration by index

```
In [11]: fruits = ["apple", "orange", "banana", "cherry"]  
  
for position in range(len(fruits)):    # by index  
    print(fruits[position])  
  
apple  
orange  
banana  
cherry
```

Since lists are mutable, it is often desirable to traverse a list, modifying each of its elements as you go.

The following code squares the numbers from 1 to 5

```
In [12]: numbers = [1, 2, 3, 4, 5]  
print(numbers)  
  
for i in range(len(numbers)):  
    numbers[i] = numbers[i] ** 2  
  
print(numbers)  
  
[1, 2, 3, 4, 5]  
[1, 4, 9, 16, 25]
```

The following code gets the maximum value from a list of integers.

```
In [13]: nums = [9, 3, 8, 11, 5, 29, 2]  
best_num = 0  
for n in nums:  
    if n > best_num:  
        best_num = n  
print(best_num)
```

29

3.7 Generate a List

Whenever you need to write a function that creates and returns a list, the pattern is usually:

```
initialize a result variable to be an empty list
loop
    create a new element
    append it to result
return the result
```

In [15]: `def doubleStuff(a_list):`

```
    new_list = []
```

```
    for value in a_list:
```

```
        new_elem = 2 * value
```

```
        new_list.append(new_elem)
```

```
    return new_list
```

```
things = [2, 5, 9]
```

```
print(things)
```

```
things = doubleStuff(things)
```

```
print(things)
```

```
[2, 5, 9]
```

```
[4, 10, 18]
```

Initialize the result variable to be an empty list

Loop

Append it to the result

3.8 Generate a List – Code challenge

Q: Write a program to generate the first 100 prime numbers. The input is the number of prime numbers, the output is a list of prime numbers.

3.9 tuple

tuple is very similar to list except that tuple is **immutable**, which means you **cannot modify** a tuple. Instead of using square brackets [] as in a list, we use parentheses () to represent tuples.

```
In [1]: colors = ('red', 'green', 'blue')
```

Once a tuple is initialized, elements cannot be changed, thus methods like append() or insert() are not applicable to tuples. But you can do indexing, slicing on tuples just like lists.

```
In [2]: colors[-1]
```

```
Out[2]: 'blue'
```

```
In [3]: colors[:-2]
```

```
Out[3]: ('red',)
```

Functions can return tuples as return values. This can be very useful.

We often want to know some batsman's highest and lowest score, or we want to find the mean and the standard deviation, or we want to know the year, the month, and the day.

In each case, a function (which can only return a single value), can create a single tuple holding multiple elements.

```
In [4]: def circleInfo(r):  
        """ Return (circumference, area) of a circle of radius r """  
        c = 2 * 3.14159 * r  
        a = 3.14159 * r * r  
        return (c, a)  
  
        print(circleInfo(10))
```

```
(62.8318, 314.159)
```


3.10 Dictionary

Dictionaries are Python's built-in mapping type. A map is an **unordered**, associative collection. The association, or mapping, is from a **key**, which can be any **immutable** type, to a **value**, which can be **any Python data object**.

Dictionary is **mutable**.

The dictionary is denoted **{key: value,...}**. Empty dictionary can be denoted **{}**

It is efficient to look up values by keys.

```
In [1]: ages = {  
        'Elbert': 23,  
        'Bob': 55,  
        'David': 8,  
        'Alice': 20,  
        'Frank': 14,  
        'Calvin': 42,  
    }  
    ages['Calvin']
```

```
Out[1]: 42
```

Add new key-value pairs into a dict.

```
In [2]: ages['Harry'] = 19  
ages
```

```
Out[2]: {'Elbert': 23,  
        'Bob': 55,  
        'David': 8,  
        'Alice': 20,  
        'Frank': 14,  
        'Calvin': 42,  
        'Harry': 19}
```

Assign value to an existing key replaces old one

```
In [3]: ages['Alice'] = 30  
ages
```

```
Out[3]: {'Elbert': 23,  
        'Bob': 55,  
        'David': 8,  
        'Alice': 30,  
        'Frank': 14,  
        'Calvin': 42,  
        'Harry': 19}
```

3.11 Dictionary method

keys(): Return the keys in the dict.

```
In [4]: inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}

for akey in inventory.keys():    # the order in which we get the keys is not defined
    print("Got key", akey, "which maps to value", inventory[akey])

ks = list(inventory.keys())
print(ks)

Got key apples which maps to value 430
Got key bananas which maps to value 312
Got key oranges which maps to value 525
Got key pears which maps to value 217
['apples', 'bananas', 'oranges', 'pears']
```

items(): Returns a view of the key-value pairs as tuples in the dict.

```
In [9]: inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}
print(list(inventory.items()))

for (k,v) in inventory.items():
    print("Got", k, "that maps to", v)

[('apples', 430), ('bananas', 312), ('oranges', 525), ('pears', 217)]
Got apples that maps to 430
Got bananas that maps to 312
Got oranges that maps to 525
Got pears that maps to 217
```

values(): Return the values in the dict.

```
In [8]: inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}

print(list(inventory.values()))

[430, 312, 525, 217]
```

get(): Return the value associated with key; None otherwise

```
In [10]: print(inventory.get('apples'))
print(inventory.get('mango'))

430
None
```

3.12 Dictionary – Code challenge

Q: Write a program called `alice_words.py` that creates a text file named `alice_words.txt` containing an alphabetical listing of all the words, and the number of times each occurs, in the text version of Alice's Adventures in Wonderland. (You can obtain a free plain text version of the book, along with many others, from <http://www.gutenberg.org>.) The first 10 lines of your output file should look something like this

Word	Count
a	631
a-piece	1
abide	1
able	1
about	94
above	3

3.13 Set

set is like a dict without values. Since keys are always unique, there is no duplicate in a set. You can also construct a set by passing a list to the set() function.

```
In [1]: s = {2, 1, 1, 3, 2}
print(s)
print(s[0]) # error, because the items in a set have no order

{1, 2, 3}
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-1-7a2a240ba0f6> in <module>
      1 s = {2, 1, 1, 3, 2}
      2 print(s)
----> 3 print(s[0]) # error, because the items in a set have no order

TypeError: 'set' object is not subscriptable
```

```
In [2]: s = set([2, 1, 1, 3, 2])
print(s)

{1, 2, 3}
```

3.14 Set method

add(): Add an element to the set.

```
In [1]: s = set([2, 1, 1, 3, 2])  
s.add(9)  
print(s)  
  
{1, 2, 3, 9}
```

remove(): Remove a specified element.

```
In [2]: s.remove(2)  
print(s)  
  
{1, 3, 9}
```

difference(): Returns a set containing the difference between sets.

```
In [3]: x = {"apple", "banana", "cherry"}  
y = {"google", "microsoft", "apple"}  
z = y.difference(x)  
  
print(z)  
  
{'microsoft', 'google'}
```

update(): updates the current set by adding items from another set.

```
In [4]: x = {"apple", "banana", "cherry"}  
y = {"google", "microsoft", "apple"}  
  
x.update(y)  
  
print(x)  
  
{'microsoft', 'banana', 'apple', 'google', 'cherry'}
```

4.1 Python Package

It makes a Python program unmaintainable if there are too many variables and functions in a single file. Through grouping functions into different modules, each file can have fewer codes. In Python, each .py file can be regarded as a module

In order to prevent conflicts between modules, Python provides **package** to organize modules.

```
packageone
├── __init__.py
└── mod.py
```

```
packagetwo
├── __init__.py
└── mod.py
```


There are two modules, and both are called mod, but if they are in different packages, *packageone* and *packagetwo*, then it is totally fine.

They can be identified as **packageone.mod** and **packagetwo.mod**

4.2 Install and Import from package

Install a package using pip

Use the syntax “pip install <package>” in the command lines

 Command Prompt

```
Microsoft Windows [Version 10.0.19041.804]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\jszhang>pip install tensorflow
```

Install a package in Jupyter

Use the syntax below in Jupyter notebook or command lines

```
In [1]: import sys
!{sys.executable} -m pip install pandas
```

Import a package

Simply import it at the beginning of your codes

```
In [2]: import math

result = math.factorial(5)
print(result)

120
```

You can import a specific part of a module

```
In [3]: from math import sqrt

Out[3]: 3.0
```

You can import and assign alias for easier recall.

```
In [5]: import numpy as np

np.average([1,2,3,4])

Out[5]: 2.5
```

4.3 Pandas overview

Pandas is an open-source Python library for performing highly specialized data analysis

- It provides a single library for data analyst to easily process data, extract data and manipulate data.
- Pandas provides two new data structures: **Series** and **DataFrame**.
- The new data structures provide data manipulation capability equivalent to SQL-based relational database within Python

```
In [1]: import pandas as pd
```

```
s = pd.Series([1, 3, 5, 4, 6, 8])  
print(s)
```

```
0    1  
1    3  
2    5  
3    4  
4    6  
5    8  
dtype: int64
```

Series

	apples
0	3
1	2
2	0
3	1

Series

	oranges
0	0
1	3
2	7
3	2

+

=

DataFrame

	apples	oranges
0	3	0
1	2	3
2	0	7
3	1	2

```
In [2]:
```

```
import numpy as np  
import pandas as pd  
df = pd.DataFrame(  
    {  
        "one": pd.Series(np.random.randn(3), index=["a", "b", "c"]),  
        "two": pd.Series(np.random.randn(4), index=["a", "b", "c", "d"])  
    }  
)  
  
print(df)
```

```
      one      two  
a  1.195455 -0.765270  
b   0.296938  0.410411  
c  -0.288035 -1.309089  
d         NaN -0.133508
```


4.4 View data

We can get the dataframe top/bottom rows information using **head()**, **tail()**.

```
In [3]: df.head() #default top 5 rows
```

Out[3]:

	A	B	C	D
0	0.165795	1.106981	-0.447365	0.439640
1	1.503472	-0.075039	-0.627865	0.563749
2	-0.430875	0.318811	0.698405	0.208957
3	0.788778	1.097461	2.289527	-0.556442
4	1.287247	1.718817	0.321562	0.408608

```
In [4]: df.tail(2)
```

Out[4]:

	A	B	C	D
4	1.287247	1.718817	0.321562	0.408608
5	1.522155	-0.320339	0.021886	0.242109

We can view the data frame index and columns, data types, data shape in following methods

```
# index
print(df.index)

# column names
print(df.columns)

# data types
print(df.dtypes)

# shape
print(df.shape)
```

It is advisable to check these properties before you do the data analysis. It gives you ideas what kind of data frame you are dealing with.

4.5 Slicing & Indexing

There are different ways to do [slicing & indexing](#). It will be used a lot to get and set subsets of pandas objects.

- `[]`: indexing with `[]` is selecting out lower-dimensional slices.
- `.loc`: `.loc` is primarily label based, but may also be used with a boolean array. `.loc` will raise `KeyError` when the items are not found.
 1. A single label, e.g. `5` or `'a'` (Note that `5` is interpreted as a label of the index. This use is not an integer position along the index.).
 2. A list or array of labels `['a', 'b', 'c']`.
 3. A slice object with labels `'a':'f'` (Note that contrary to usual Python slices, both the start and the stop are included, when present in the index! See Slicing with labels and Endpoints are inclusive.)
 4. A boolean array (any NA values will be treated as False).
 5. A callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above)
- `.iloc`: `.iloc` is primarily integer position based (from 0 to length-1 of the axis), but may also be used with a boolean array. `.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except slice indexers which allow out-of-bounds indexing.
 1. An integer e.g. `5`.
 2. A list or array of integers `[4, 3, 0]`.
 3. A slice object with ints `1:7`.
 4. A boolean array (any NA values will be treated as False).
 5. A callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above).
- `.at`: similarly to `.loc`, `.at` provides label based scalar lookups
- `.iat`: similarly to `.iloc`, `.iat` provides integer based scalar lookups
- `.isin`: we can select rows from a DataFrame using a boolean vector. `.isin()` method can return a boolean vector.
- `.where`: It is similar to `.isin` method. `.where` can return a boolean vector.

4.5 Slicing & Indexing []

Indexing with [] is selecting out lower-dimensional slices.

```
df = pd.DataFrame(np.random.randn(6, 4), columns=list("ABCD"))  
print(df)
```

	A	B	C	D
0	0.165795	1.106981	-0.447365	0.439640
1	1.503472	-0.075039	-0.627865	0.563749
2	-0.430875	0.318811	0.698405	0.208957
3	0.788778	1.097461	2.289527	-0.556442
4	1.287247	1.718817	0.321562	0.408608
5	1.522155	-0.320339	0.021886	0.242109

```
df["A"] # select the column A
```

```
0    0.165795  
1    1.503472  
2   -0.430875  
3    0.788778  
4    1.287247  
5    1.522155  
Name: A, dtype: float64
```

```
df[0:2] # select the first 2 rows
```

	A	B	C	D
0	0.165795	1.106981	-0.447365	0.439640
1	1.503472	-0.075039	-0.627865	0.563749

```
df[['C', 'D']]
```

	C	D
0	-0.447365	0.439640
1	-0.627865	0.563749
2	0.698405	0.208957
3	2.289527	-0.556442
4	0.321562	0.408608
5	0.021886	0.242109

4.5 Slicing & Indexing loc

.loc is primarily label based.

1. A single label, e.g. `5` or `'a'` (Note that 5 is interpreted as a label of the index. This use is not an integer position along the index.).
2. A list or array of labels `['a', 'b', 'c']`.
3. A slice object with labels `'a': 'f'` (Note that contrary to usual Python slices, both the start and the stop are included, when present in the index! See Slicing with labels and Endpoints are inclusive.)
4. A boolean array (any NA values will be treated as False).
5. A callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above)

```
df2 = pd.DataFrame(  
    {  
        "A": 1.0,  
        "B": pd.Timestamp("20210402"),  
        "C": pd.Series(1, index=list(range(4)), dtype="float32"),  
        "D": np.array([3] * 4, dtype="int32"),  
        "E": pd.Categorical(["test", "train", "test", "train"]),  
        "F": "foo",  
    }  
)  
print(df2)
```

	A	B	C	D	E	F
0	1.0	2021-04-02	1.0	3	test	foo
1	1.0	2021-04-02	1.0	3	train	foo
2	1.0	2021-04-02	1.0	3	test	foo
3	1.0	2021-04-02	1.0	3	train	foo

```
df2.loc[1:2, ["A", "B"]] # the 2nd and 3rd row in column A and column B
```

	A	B
1	1.0	2021-04-02
2	1.0	2021-04-02

```
df2.loc[:, ["A", "B"]] # select all rows in column A, column B
```

	A	B
0	1.0	2021-04-02
1	1.0	2021-04-02
2	1.0	2021-04-02
3	1.0	2021-04-02

4.5 Slicing & Indexing iloc

.iloc is primarily integer position based

1. An integer e.g. `5`.
2. A list or array of integers `[4, 3, 0]`.
3. A slice object with ints `1:7`.
4. A boolean array (any NA values will be treated as False).
5. A callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above).

```
df2 = pd.DataFrame(  
    {  
        "A": 1.0,  
        "B": pd.Timestamp("20210402"),  
        "C": pd.Series(1, index=list(range(4)), dtype="float32"),  
        "D": np.array([3] * 4, dtype="int32"),  
        "E": pd.Categorical(["test", "train", "test", "train"]),  
        "F": "foo",  
    }  
)  
print(df2)
```

	A	B	C	D	E	F
0	1.0	2021-04-02	1.0	3	test	foo
1	1.0	2021-04-02	1.0	3	train	foo
2	1.0	2021-04-02	1.0	3	test	foo
3	1.0	2021-04-02	1.0	3	train	foo

`df2.iloc[3] # the 4th row`

```
A      1  
B    2021-04-02 00:00:00  
C      1  
D      3  
E     train  
F      foo  
Name: 3, dtype: object
```

`df2.iloc[1:3, :] # 2nd and 3rd row in all columns`

	A	B	C	D	E	F
1	1.0	2021-04-02	1.0	3	train	foo
2	1.0	2021-04-02	1.0	3	test	foo

4.5 Slicing & Indexing boolean vector

```
df = pd.DataFrame(np.random.randn(6, 4), columns=list("ABCD"))  
print(df)
```

	A	B	C	D
0	0.165795	1.106981	-0.447365	0.439640
1	1.503472	-0.075039	-0.627865	0.563749
2	-0.430875	0.318811	0.698405	0.208957
3	0.788778	1.097461	2.289527	-0.556442
4	1.287247	1.718817	0.321562	0.408608
5	1.522155	-0.320339	0.021886	0.242109

```
df[df["A"] > 0] # use boolean index to filter the dataframe
```

	A	B	C	D
0	0.165795	1.106981	-0.447365	0.439640
1	1.503472	-0.075039	-0.627865	0.563749
3	0.788778	1.097461	2.289527	-0.556442
4	1.287247	1.718817	0.321562	0.408608
5	1.522155	-0.320339	0.021886	0.242109

```
In [18]: df["E"] = ["one", "one", "two", "three", "four", "three"] # new column is added to the dataframe.
```

```
In [20]: df
```

```
Out[20]:
```

	A	B	C	D	E
0	0.165795	1.106981	-0.447365	0.439640	one
1	1.503472	-0.075039	-0.627865	0.563749	one
2	-0.430875	0.318811	0.698405	0.208957	two
3	0.788778	1.097461	2.289527	-0.556442	three
4	1.287247	1.718817	0.321562	0.408608	four
5	1.522155	-0.320339	0.021886	0.242109	three

```
In [19]: df[df["E"].isin(["two", "four"])]
```

```
Out[19]:
```

	A	B	C	D	E
2	-0.430875	0.318811	0.698405	0.208957	two
4	1.287247	1.718817	0.321562	0.408608	four

```
In [21]: df.where(df['B'] > 0) # which is equal to df[df['B'] < 0]
```

```
Out[21]:
```

	A	B	C	D	E
0	0.165795	1.106981	-0.447365	0.439640	one
1	NaN	NaN	NaN	NaN	NaN
2	-0.430875	0.318811	0.698405	0.208957	two
3	0.788778	1.097461	2.289527	-0.556442	three
4	1.287247	1.718817	0.321562	0.408608	four
5	NaN	NaN	NaN	NaN	NaN

4.6 Operations

- `sum()` : Aggregation function that gives the total sum of a column
- `mean()` : Aggregation function that gives the average value of a column
- `std()` : Aggregation function that gives the standard deviation value of a column
- `count()` : Counts the number of fill rows in the columns, where empty rows are ignored.
- `unique()` : Similar to `set()`, but only works for Series object, i.e `data['Age']`
- `nunique()` : Much like `unique()`, except it counts the number of unique elements in the Series.
- `value_counts()` : Counting duplicated values

```
df = pd.DataFrame(np.random.randn(6, 4), columns=list("ABCD"))  
print(df)
```

	A	B	C	D
0	0.165795	1.106981	-0.447365	0.439640
1	1.503472	-0.075039	-0.627865	0.563749
2	-0.430875	0.318811	0.698405	0.208957
3	0.788778	1.097461	2.289527	-0.556442
4	1.287247	1.718817	0.321562	0.408608
5	1.522155	-0.320339	0.021886	0.242109

`df.mean()` # get mean value

```
A    0.806095  
B    0.641115  
C    0.376025  
D    0.217770  
dtype: float64
```

`df['E'].nunique()`

```
4
```

`df['E'].value_counts()`

```
three    2  
one      2  
two      1  
four     1  
Name: E, dtype: int64
```

`df[['A','B','C','D']].apply(lambda x: x.max() - x.min())`

```
A    1.953029  
B    2.039156  
C    2.917391  
D    1.120191  
dtype: float64
```

4.7 Merge Data

Join Types:

INNER: Returns records that have matching values in both tables

LEFT: Returns all records from the left table, and the matched records from the right table

RIGHT: Returns all records from the right table, and the matched records from the left table

OUTER: Returns all records when there is a match in either left or right table

Table 1

Name	Gender
Andy	M
Bob	M
Charlie	M
Elsa	F
Felicia	F

```
df_inner = pd.merge(df_1, df_2, on = "Name", how = "inner")  
print(df_inner)
```

```
# left join
```

```
df_left = pd.merge(df_1, df_2, on = "Name", how = "left")  
print(df_left)
```

```
# right join
```

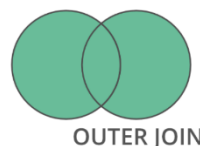
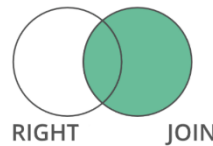
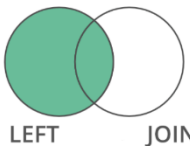
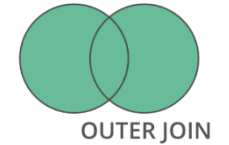
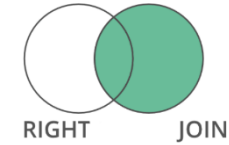
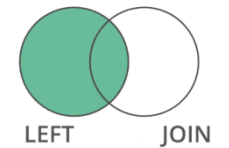
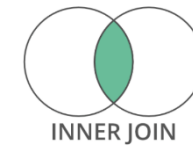
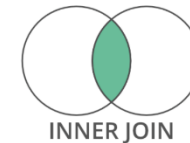
```
df_right = pd.merge(df_1, df_2, on = "Name", how = "right")  
print(df_right)
```

```
# outer join
```

```
df_outer = pd.merge(df_1, df_2, on = "Name", how = "outer")  
print(df_outer)
```

Table 1

Name	Gender
Andy	M
Bob	M
Charlie	M
Elsa	F
Felicia	F



Name	Gender	Age
Andy	M	15
Bob	M	23
Elsa	F	18


Name	Gender	Age
Andy	M	15
Bob	M	23
Charlie	M	N/A
Elsa	F	18
Felicia	F	N/A

Name	Age	Gender
Andy	15	M
Bob	23	M
Damien	27	N/A
Elsa	18	18
Gertrude	46	N/A

Name	Gender	Age
Andy	M	15
Bob	M	23
Charlie	M	N/A
Damien	N/A	27
Elsa	F	18
Felicia	F	N/A
Gertrude	N/A	46

4.8 Concat Data

```
df1 = pd.DataFrame(  
    {  
        "A": ["A0", "A1", "A2", "A3"],  
        "B": ["B0", "B1", "B2", "B3"],  
        "C": ["C0", "C1", "C2", "C3"],  
        "D": ["D0", "D1", "D2", "D3"],  
    },  
    index=[0, 1, 2, 3],  
)  
  
df2 = pd.DataFrame(  
    {  
        "A": ["A0", "A5", "A6", "A7"],  
        "B": ["B0", "B5", "B6", "B7"],  
        "C": ["C0", "C5", "C6", "C7"],  
        "D": ["D0", "D5", "D6", "D7"],  
    },  
    index=[4, 5, 6, 7],  
)  
  
df3 = pd.DataFrame(  
    {  
        "A": ["A0", "A9", "A10", "A11"],  
        "B": ["B0", "B9", "B10", "B11"],  
        "C": ["C0", "C9", "C10", "C11"],  
        "D": ["D0", "D9", "D10", "D11"],  
    },  
    index=[8, 9, 10, 11],  
)  
  
frames = [df1, df2, df3]  
result = pd.concat(frames)
```



	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Note

It can be useful when we want to combine multiple files into one single file for our data analysis.

```
frames = [ process_file(f) for f in files ]  
result = pd.concat(frames)
```

4.9 Group Data

In data analysis, we often need to answer business questions by summarizing data. For example,

- "What is the department sales in Q1?"
- "How many new customers we have acquired in the past year?"
- "Which product saw the biggest increase during the past 6 months?"

To answer these questions, we need to group the data by certain dimension(s) and calculate the metrics.

By “group by” we are referring to a process involving one or more of the following steps:

- **Splitting** the data into groups based on some criteria
- **Applying** a function to each group independently
- **Combining** the results into a data structure

```
df = pd.DataFrame(  
    {  
        "A": ["foo", "bar", "foo", "bar", "foo", "bar", "foo", "foo"],  
        "B": ["one", "one", "two", "three", "two", "two", "one", "three"],  
        "C": np.random.randn(8),  
        "D": np.random.randn(8),  
    })  
df
```

	A	B	C	D
0	foo	one	-0.557548	0.418605
1	bar	one	-1.381606	0.351290
2	foo	two	2.183459	0.548590
3	bar	three	1.693160	0.603128
4	foo	two	-0.987693	-0.934828
5	bar	two	0.196928	0.586793
6	foo	one	-0.853758	0.026537
7	foo	three	1.932689	-2.028865

```
df.groupby(["A", "B"]).sum()
```

		C	D
bar	one	-1.381606	0.351290
	three	1.693160	0.603128
	two	0.196928	0.586793
foo	one	-1.411307	0.445142
	three	1.932689	-2.028865
	two	1.195766	-0.386238

```
df.groupby(['A']).agg({'C': 'mean', 'D': 'sum'})
```

	C	D
bar	0.169494	1.541211
foo	0.343430	-1.969961

4.10 Get Data In/Out

We can use pandas to read files to extract data. pandas supports a wide range of file format, such as csv, excel,json, HTML, SQL and so on.

Format Type	Data Description	Reader	Writer
text	CSV	<code>read_csv</code>	<code>to_csv</code>
text	Fixed-Width Text File	<code>read_fwf</code>	
text	JSON	<code>read_json</code>	<code>to_json</code>
text	HTML	<code>read_html</code>	<code>to_html</code>
text	Local clipboard	<code>read_clipboard</code>	<code>to_clipboard</code>
binary	MS Excel	<code>read_excel</code>	<code>to_excel</code>
binary	OpenDocument	<code>read_excel</code>	
binary	HDF5 Format	<code>read_hdf</code>	<code>to_hdf</code>
binary	Feather Format	<code>read_feather</code>	<code>to_feather</code>
binary	Parquet Format	<code>read_parquet</code>	<code>to_parquet</code>
binary	ORC Format	<code>read_orc</code>	
binary	Msgpack	<code>read_msgpack</code>	<code>to_msgpack</code>
binary	Stata	<code>read_stata</code>	<code>to_stata</code>
binary	SAS	<code>read_sas</code>	
binary	SPSS	<code>read_spss</code>	
binary	Python Pickle Format	<code>read_pickle</code>	<code>to_pickle</code>
SQL	SQL	<code>read_sql</code>	<code>to_sql</code>
SQL	Google BigQuery	<code>read_gbq</code>	<code>to_gbq</code>

Read csv

```
df = pd.read_csv("data/world-happiness-report.csv")
```

Write to csv

```
top_10_GDP.to_csv('data/top10gdp_country.csv')
```

Read excel

```
apple_price = pd.read_excel('data/AAPL_excel.xls', sheet_name= 'AAPL')
apple_price.head()
```

Read files matching a pattern

```
import glob
import os

files = glob.glob("data/file_*.csv")
result = pd.concat([pd.read_csv(f) for f in files], ignore_index=True)

result
```

4.11 Handling Missing Values

Before you start cleaning a data set, it's good to get a general feel for the data. What are the features?

- What are the expected types (int, float, string, Boolean)?
- Is there obvious missing data (values that Pandas can detect)?
- Is there other types of missing data that's not so obvious (can't easily detect with Pandas)?

Sources of Missing Values, such as user forgot to fill in a field; there was a programming error etc.

```
import pandas as pd
```

```
# Read csv file into a pandas dataframe  
df = pd.read_csv("data/property_data.csv")
```

```
# Take a Look at df  
df
```

	PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
0	100001000.0	104.0	PUTNAM	Y	3	1	1000
1	100002000.0	197.0	LEXINGTON	N	3	1.5	--
2	100003000.0	NaN	LEXINGTON	N	NaN	1	850
3	100004000.0	201.0	BERKELEY	12	1	NaN	700
4	NaN	203.0	BERKELEY	Y	3	2	1600
5	100006000.0	207.0	BERKELEY	Y	NaN	1	800
6	100007000.0	NaN	WASHINGTON	NaN	2	HURLEY	950
7	100008000.0	213.0	TREMONT	Y	1	1	NaN
8	100009000.0	215.0	TREMONT	Y	na	2	1800

① Drop NA values

```
df.dropna(how="any") #To drop any rows that have missing data.
```

	PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
0	100001000.0	104.0	PUTNAM	Y	3	1	1000
1	100002000.0	197.0	LEXINGTON	N	3	1.5	--
8	100009000.0	215.0	TREMONT	Y	na	2	1800

```
# Replace using median  
median = df['ST_NUM'].median()  
df['ST_NUM'].fillna(median, inplace=True)  
  
df
```

	PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
0	100001000.0	104.0	PUTNAM	Y	3	1	1000
1	100002000.0	197.0	LEXINGTON	N	3	1.5	--
2	100003000.0	125.0	LEXINGTON	N	NaN	1	850
3	100004000.0	201.0	BERKELEY	12	1	NaN	700
4	NaN	203.0	BERKELEY	Y	3	2	1600
5	100006000.0	207.0	BERKELEY	Y	NaN	1	800
6	100007000.0	125.0	WASHINGTON	NaN	2	HURLEY	950
7	100008000.0	213.0	TREMONT	Y	1	1	NaN
8	100009000.0	215.0	TREMONT	Y	na	2	1800

② Impute values

5.1 Numpy Introduction

Numpy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

```
import numpy as np
a = np.arange(15).reshape(3, 5)
print(a)

print(a.shape)

print(a.ndim)

print(a.dtype.name)

print(a.size)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
(3, 5)
2
int32
15
```

- NumPy provides a multi-dimensional array data structure, **ndarray**, which is a fast, flexible container for large data sets in Python.
- In an ndarray, all the elements must be the **same** type.
- Every ndarray has a **shape**, which is a tuple indicating the size of each dimension, and a **dtype**, describing the data type of the elements in the array.

5.1 Array Creation

The most common way to create a **ndarray** is through a regular Python list or tuple using array function:

```
import numpy as np
data = [1,2,3]
a = np.array(data)
print(a)
a.dtype
```

```
[1 2 3]
```

```
dtype('int32')
```

```
arr = np.array([(2.1, 4, 7.0), (-5, 72, 4.3)])
arr
```

```
array([[ 2.1,  4. ,  7. ],
       [-5. , 72. ,  4.3]])
```

```
np.zeros( (5,3) )
```

```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

```
np.ones( (2, 4) )
```

```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

- Be careful that the input argument is a list.

```
a = np.array(1,2,3,4)
```

```
-----
TypeError
```

```
Traceback (most recent call last)
```

```
<ipython-input-3-dfad7faaf733> in <module>
```

```
----> 1 a = np.array(1,2,3,4)
```

```
TypeError: array() takes from 1 to 2 positional arguments but 4 were given
```

- array transforms sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three-dimensional arrays, and so on.
- NumPy offers several functions to create arrays with initial placeholder content.
- The function zeros creates an array full of zeros.
- The function ones creates an array full of one.
- The function empty creates an array full of random float64 values.

5.1 Array Creation

```
np.arange(8)
```

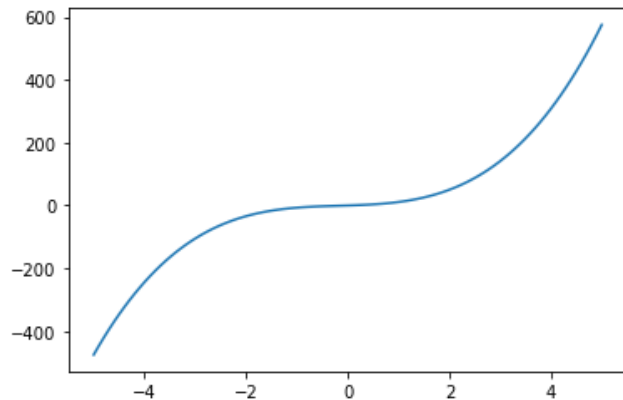
```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

```
np.linspace(0, 18, 10) # 10 numbers from 0 to 18
```

```
array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14., 16., 18.])
```

```
import matplotlib.pyplot as plt

x_ = np.linspace(-5, 5, 100)
y_ = 4 * (x_**3) + 2 * (x_**2) + 5 * x_
plt.plot(x_, y_)
plt.show()
```



- To create sequences of numbers, NumPy provides the **arange** function which is analogous to the Python built-in **range**, but returns an array.
- It's usually not possible to predict the number of elements obtained, due to the finite floating-point precision. For this reason, it is better to use the function **linspace** that receives as an argument the number of elements that we want.
- **linspace** functions can be useful to evaluate function at lots of points

5.2 Array print

One-dimensional arrays are then printed as rows, bi-dimensionals as matrices and tri-dimensionals as lists of matrices. The **reshape** function returns its argument with a modified shape.

```
a = np.arange(6) # 1d array
print(a)
```

```
[0 1 2 3 4 5]
```

```
b = np.arange(12).reshape(4,3) # 2d array
print(b)
```

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```

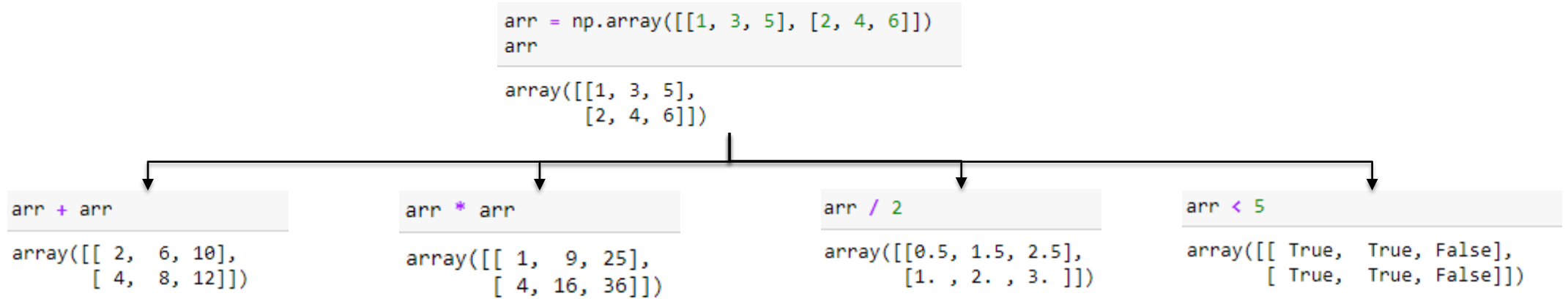
```
c = np.arange(24).reshape(2,3,4) # 3d array
print(c)
```

```
[[[ 0  1  2  3]
   [ 4  5  6  7]
   [ 8  9 10 11]]
```

```
 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```


5.3 Array operations

Arithmetic operators on arrays apply **elementwise**. A new array is created and filled with the result.



The **matrix product** can be performed using the **@** operator (in python ≥ 3.5) or the dot function

```
X = np.array( [ [2, 3], # [ [a,b],
                 [0, 1] ] # [c,d] ]
Y = np.array( [ [1, 2], # [ [e,f],
                 [3, 4] ] # [g,h] ]
print(X @ Y)
print(X.dot(Y))

#[ [a*e+b*g, a*f+b*h]
#  [c*e+d*g, c*f+d*h] ]

[[11 16]
 [ 3  4]]
[[11 16]
 [ 3  4]]
```

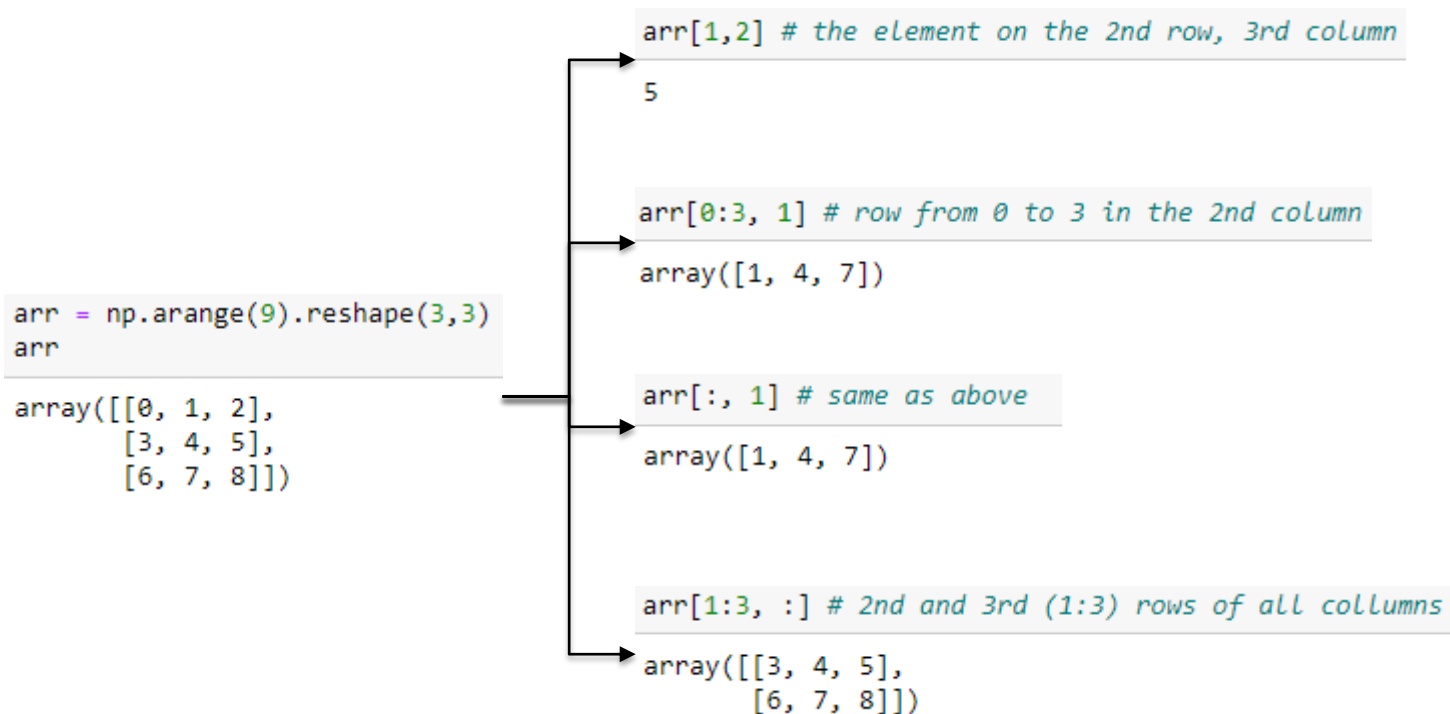
Matrices Multiplication

$$\begin{bmatrix} 4 & 2 & 4 \\ 8 & 3 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 & 5 \\ 2 & 8 \\ 7 & 9 \end{bmatrix} = \begin{bmatrix} 44 & 72 \\ 37 & 73 \end{bmatrix}$$

→ $4 \times 3 + 2 \times 2 + 4 \times 7 = 44$
→ $4 \times 5 + 2 \times 8 + 4 \times 9 = 72$
→ $8 \times 3 + 3 \times 2 + 1 \times 7 = 37$
→ $8 \times 5 + 3 \times 8 + 1 \times 9 = 73$

5.4 Array indexing & slicing

One-dimensional arrays can be indexed, sliced and iterated over, much like lists.



What's the following slicing result on a 3*3 array?

```
arr[:2, 1:]
```

```
arr[2] arr[2, :] arr[2:, :]
```

```
arr[:, :2]
```

```
arr[1, :2] arr[1:2, :2]
```

5.5 Array indexing tricks

In addition to indexing by integers and slices, arrays can be indexed by **arrays of integers** and **arrays of Booleans**.

Indexing with Arrays of Indices

```
a = np.arange(12)**2 # the first 12 square numbers
i = np.array([1, 1, 3, 8, 5]) # an array of indices
print(a)
```

```
a[i]

[ 0  1  4  9 16 25 36 49 64 81 100 121]
array([ 1,  1,  9, 64, 25], dtype=int32)
```

```
j = np.array([[3, 4], [9, 7]]) # a bidimensional array of indices
a[j]
```

```
array([[ 9, 16],
       [81, 49]], dtype=int32)
```

```
palette = np.array([[0, 0, 0],      # black
                    [255, 0, 0],    # red
                    [0, 255, 0],     # green
                    [0, 0, 255],     # blue
                    [255, 255, 255]]) # white
```

```
image = np.array([[0, 1, 2, 0],
                  [0, 3, 4, 0]])
# each value corresponds to a color in the palette
```

```
palette[image] # the (2, 4, 3) color image
```

```
array([[ [ 0,  0,  0],
         [255,  0,  0],
         [ 0, 255,  0],
         [ 0,  0,  0]],

       [[ [ 0,  0,  0],
         [ 0,  0, 255],
         [255, 255, 255],
         [ 0,  0,  0]])])
```

The example converts an image of labels into a color image using a palette.

5.5 Array indexing tricks

In addition to indexing by integers and slices, arrays can be indexed by **arrays of integers** and **arrays of Booleans**.

Indexing with Boolean Arrays

```
a = np.arange(12).reshape(3,4)
b = a > 4
b # b is a boolean with a's shape

array([[False, False, False, False],
       [False,  True,  True,  True],
       [ True,  True,  True,  True]])
```

```
a[b] # 1d array with the selected elements

array([ 5,  6,  7,  8,  9, 10, 11])
```

Indexing with Boolean array can be very useful in assignments

```
a[b] = 0
a

array([[0, 1, 2, 3],
       [4, 0, 0, 0],
       [0, 0, 0, 0]])
```

5.6 Numpy methods

.random.random

return random floats in the half-open interval [0.0, 1.0).

```
np.random.random(5)

array([0.26166948, 0.52634598, 0.11630677, 0.27079236, 0.1666527 ])
```

.floor

return the floor of the input, element-wise.

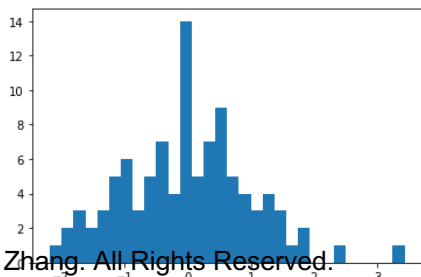
```
arr = np.array([-2.5, -1.7, -2.0, 0.2, 1.3])
np.floor(arr)

array([-3., -2., -2., 0., 1.]
```

.random.normal

returns random samples from a normal distribution.

```
import matplotlib.pyplot as plt
values = np.random.normal(size=100)
plt.hist(values, 30)
plt.show()
```



.array_split

split an array into multiple sub-arrays.

```
import numpy as np
x = np.arange(7)
np.array_split(x, 2)

[array([0, 1, 2, 3]), array([4, 5, 6])]
```

.diag

extract a diagonal or construct a diagonal array.

```
x = np.arange(9).reshape((3,3))
x

array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

np.diag(x)

array([0, 4, 8])
```

5.7 Numpy functions

Universal functions

A universal function, or **ufunc** is a function that performs elementwise operations on data in ndarray.

```
arr = np.array([3, -6, 0, 19, -5.4])
np.abs(arr)

array([ 3. ,  6. ,  0. , 19. ,  5.4])
```

```
arr1 = np.array([1,4,5,8])
arr2 = np.array([2,3,6,7])
np.maximum(arr1, arr2)

array([2, 4, 6, 8])
```

```
arr = np.arange(10)
arr.sum()

45
```

Unary ufuncs

```
sqrt
square
exp
log, log10, log2
ceil
floor
```

Binary ufuncs

```
subtract
multiply
divide, floor_divide
power

maximum
```

Mathematical and Statistical Methods

```
.mean
.std, .var
.min, .max
.argmax, .argmin
```

Indices of minimum and maximum elements, respectively.

```
.cumsum
```

Cumulative sum of elements starting from 0.

```
.cumprod
```

Cumulative product of elements starting from 1.

5.8 Data processing using arrays

Where:

creates a new array of values base on another array.

```
arr = np.random.randn(3,3)
arr
```

```
array([[ 0.66475571, -1.67258499, -0.68724068],
       [-1.14677306, -0.74840783, -0.88483543],
       [ 1.08542353,  0.12343565,  0.18753589]])
```

```
# if the value less than 0, then 0, otherwise, unchanged
np.where(arr < 0, 0, arr)
```

```
array([[0.66475571, 0.        , 0.        ],
       [0.        , 0.        , 0.        ],
       [1.08542353, 0.12343565, 0.18753589]])
```

any and **all** are two methods used in Boolean arrays.
any checks if one or more values in an array is **True**.
all checks if every value in an array is **True**.

```
bools = np.array([True, False, True, True])
```

```
print( bools.any() )
print( bools.all() )
```

```
True
False
```

sum is often used to count **True** values in a Boolean array.

```
(arr > 0).sum()
```

```
4
```

5.9 Numpy – Code challenge

Q: Write a program to solve sudoku.

We represent a sudoku in a 2-D array. The empty cell is filled with zero.

```
puzzle = [[5,3,0,0,7,0,0,0,0],  
          [6,0,0,1,9,5,0,0,0],  
          [0,9,8,0,0,0,0,6,0],  
          [8,0,0,0,6,0,0,0,3],  
          [4,0,0,8,0,3,0,0,1],  
          [7,0,0,0,2,0,0,0,6],  
          [0,6,0,0,0,0,2,8,0],  
          [0,0,0,4,1,9,0,0,5],  
          [0,0,0,0,8,0,0,7,9]]
```


6.1 Functional Programming

Python's functional programming features make data processing more convenient.

Lambda function

A lambda function is a small anonymous function. A lambda function can take any number of arguments but can only have one expression.

```
In [1]: x = lambda a, b, c : a + b + c  
        print(x(5, 6, 2))
```

13

Filter function

A filter function returns an iterator where the items are filtered through a function to test if the item is accepted or not.

```
In [2]: ages = [5, 12, 17, 18, 24, 32]  
  
def myFunc(x):  
    if x < 18:  
        return False  
    else:  
        return True  
  
adults = filter(myFunc, ages)  
  
for x in adults:  
    print(x)
```

18
24
32

6.2 Functional Programming

Python's functional programming features make data processing more convenient.

Map function

A map function executes a specified function for each item in an iterable. The item is sent to the function as a parameter.

```
In [4]: def addition(n):  
        return n + n  
  
        # We double all numbers using map()  
        numbers = (1, 2, 3, 4)  
        result = map(addition, numbers)  
        print(list(result))  
  
[2, 4, 6, 8]
```

Reduce function

A reduce function is used to apply a particular function passed in its argument to all the list elements mentioned in the sequence passed along. It is defined in **functools** module.

```
In [7]: import functools  
  
        lis = [ 1 , 3, 5, 6, 2, ]  
  
        # using reduce to compute sum of list  
        print ("The sum of the list elements is : ",end="")  
        print (functools.reduce(lambda a,b : a+b,lis))  
  
        # using reduce to compute maximum element from list  
        print ("The maximum element of the list is : ",end="")  
        print (functools.reduce(lambda a,b : a if a > b else b,lis))  
  
The sum of the list elements is : 17  
The maximum element of the list is : 6
```

6.3 Functional Programming

Map + Lambda to transform and generate a list

Map + Lambda is a convenient way to transform and generate a list.

The map() function runs a lambda function over the list [1, 2, 3, 4, 5], building a list-like collection of the results, like this:

```
In [1]: list(map(lambda n: n * 2, [1, 2, 3, 4, 5]))
```

```
Out[1]: [2, 4, 6, 8, 10]
```

lambda n: n * 2

[1, 2, 3, 4, 5]



[2, 4, 6, 8, 10]

```
In [2]: strs = ['Python', 'is', 'great']  
list(map(lambda s: s.upper() + '!', strs))
```

```
Out[2]: ['PYTHON!', 'IS!', 'GREAT!']
```

6.4 Functional Programming

Filter + Lambda to filter a list

Filter + Lambda is a convenient way to filter a list and return a subset of list where function returns true.


```
In [2]: ages = [5, 12, 17, 18, 24, 32]

def myFunc(x):
    if x < 18:
        return False
    else:
        return True

adults = filter(myFunc, ages)

for x in adults:
    print(x)
```

```
18
24
32
```



```
In [3]: ages = [5, 12, 17, 18, 24, 32]
list(filter(lambda x: x >= 18, ages))

Out[3]: [18, 24, 32]
```

7.1 Exception

Sometimes, running our syntactically correct code may cause errors, and we call it exceptions.

```
In [1]: print('Result:' + 200)
```

```
-----  
TypeError                                 Traceback (most recent call last)  
<ipython-input-1-fe454ded8bee> in <module>  
----> 1 print('Result:' + 200)  
  
TypeError: can only concatenate str (not "int") to str
```

```
In [2]: user_input = input('Input a number: ')  
        print(int(user_input))
```

Input a number: a

```
-----  
ValueError                               Traceback (most recent call last)  
<ipython-input-2-d280f79920ff> in <module>  
      1 user_input = input('Input a number: ')  
----> 2 print(int(user_input))  
  
ValueError: invalid literal for int() with base 10: 'a'
```

```
In [3]: with open('no_such_file.txt', 'r') as f:  
        print(f.read())
```

```
-----  
FileNotFoundError                       Traceback (most recent call last)  
<ipython-input-3-1554493fe29f> in <module>  
----> 1 with open('no_such_file.txt', 'r') as f:  
      2     print(f.read())
```

FileNotFoundError: [Errno 2] No such file or directory: 'no_such_file.txt'

7.2 Standard Exceptions

Language Exceptions	Description
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
ImportError	Raised when an import statement fails.
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.
NameError	Raised when an identifier is not found in the local or global namespace.
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
LookupError	Base class for all lookup errors.
IndexError	Raised when an index is not found in a sequence.
KeyError	Raised when the specified key is not found in the dictionary.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.
MemoryError	Raised when a operation runs out of memory.
RecursionError	Raised when the maximum recursion depth has been exceeded.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.

Math Exceptions	Description
ArithmeticError	Base class for all errors that occur for numeric calculation. You know a math error occurred, but you don't know the specific error.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types.
I/O Exceptions	Description
FileNotFoundError	Raised when a file or directory is requested but doesn't exist.
IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist. Also raised for operating system-related errors.
PermissionError	Raised when trying to run an operation without the adequate access rights.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.

7.3 Handling Exceptions

You can detect exceptions and handle them properly using **try-except blocks**.

Catch All Exceptions

```
In [4]: while True:
        try:
            user_input = input('Input a number: ')
            print(int(user_input))
            break
        except:
            print('[invalid number]')
            continue
```

```
Input a number: a
[invalid number]
Input a number: 10
10
```

1. Try clause between try and except is executed.
2. If no exception occurs, the except clause is skipped and execution of the try statement is finished.
3. If an exception occurs during execution of the try clause, the rest of the clause is skipped. The except clause is executed and then the execution continues after the try statement.

Catch A Specific Exception

```
In [5]: try:
        user_input = input('Input a number: ')
        print(int(user_input))
    except ValueError as error:
        print(error)
```

```
Input a number: a
invalid literal for int() with base 10: 'a'
```

You can catch a specific type of exception and define a variable for it.

7.3 Clean-up After Exceptions

The try statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances.

```
In [6]: try:
        raise KeyboardInterrupt
    finally:
        print('Goodbye, world!')
```

Goodbye, world!

```
KeyboardInterrupt                                Traceback (most recent call last)
<ipython-input-6-ca8991ac7661> in <module>
      1 try:
----> 2     raise KeyboardInterrupt
      3 finally:
      4     print('Goodbye, world!')
```

KeyboardInterrupt:

One place where you want to include exception handling is when you read or write to a file. Here is a typical example of file processing.

```
try:
    f = open("my_file.txt", "w")
    try:
        f.write("Writing some data to the file")
    finally:
        f.close()
except IOError:
    print "Error: my_file.txt does not exist or it can't be opened for output."
```


7.4 Debugging

There are different kinds of errors can occur in program. It is useful if we can distinguish them:

1. **Syntax error:** It usually indicates something wrong with syntax of your code, such as
 - Omitting colon at the end of functions
 - Wrong indentation.
 - Strings should have matching quotation marks.
 - Unclosed bracket {, (, [
 - Sign = is not the same as ==
2. **Runtime error:** The program is syntactically correct, but it did not give what we expected, such as
 - Infinite loop
 - Infinite recursion
 - Exception handling, such as NameError, TypeError, KeyError, IndexError etc.
3. **Semantic error:** Semantic errors are hard to debug because compiler and runtime system provide no information
 - Break the program into smaller components and test each component independently.
 - Get some rest before you get frustrated.

8.1 Working with Files

In real life data reside in files. File reading and writing are common input/output (IO) operations. In Python, we must **open** files before we can use them and **close** them when we are done with them.

open(): create a file object

```
In [1]: f = open('../folder/data.txt', 'r')
```

- **'folder/data.txt'** is the **path** to the file we want to read
- **'r'** stands for reading mode.
- If the file does not exist, open() will throw an **IOError** exception.

read(): read from file object and get the content.

```
In [2]: f.read()
```

close(): close the file to release system resources

```
In [3]: f.close()
```

Character	Meaning
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	open for exclusive creation, failing if the file already exists
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)
'+'	open for updating (reading and writing)

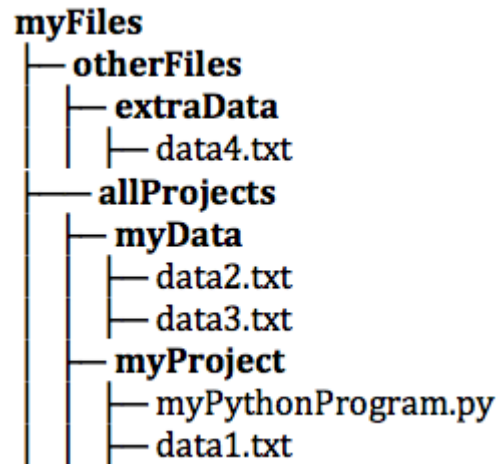
8.2 Locate Files on your disk

Files are located on disk by their **path**. You can think of **path = directory + filename**. For example,

In windows, the path might be **C:\Users\yourname\My Documents\hello.txt**

In Mac, the path might be **/Users/yourname/hello.txt**

A file hierarchy contains directories which contains files and other sub-directories.



myPythonProgram.py could access data files using the following relative file paths:

- **data1.txt**
- **../myData/data2.txt**
- **../myData/data3.txt**
- **../../otherFiles/extraData/data4.txt**

To specify a parent folder, we can use special **..** notation.

It can be used multiple times to move multiple levels up a file hierarchy

If your file and your Python program are in different directories, then you must refer to one or more directories.

In a **relative file path** to the file like this: **open('../myData/data3.txt', 'r')**, or

In an **absolute file path** like **open('/users/yourname/myFiles/allProjects/myData/data3.txt', 'r')**.

8.3 Read file content

We have a file called “graduate course type in Singapore.txt”.

We will **read** each line of the file and print it with some additional text.

Because text files are sequences of lines of text, we can use the **for loop** to iterate through each line of the file.

```
1993,Males,Education,na
1993,Males,Applied Arts,na
1993,Males,Humanities & Social Sciences,481
1993,Males,Mass Communication,na
1993,Males,Accountancy,295
1993,Males,Business & Administration,282
1993,Males,Law,92
1993,Males,"Natural, Physical & Mathematical Sciences",404
1993,Males,Medicine,95
1993,Males,Dentistry,14
1993,Males,Health Sciences,10
1993,Males,Information Technology,264
1993,Males,Architecture & Building,132
1993,Males,Engineering Sciences,1496
1993,Males,Services,na
1993,Females,Education,na
1993,Females,Applied Arts,na
1993,Females,Humanities & Social Sciences,1173
1993,Females,Mass Communication,na
1993,Females,Accountancy,396
1993,Females,Business & Administration,708
```

A **line** of a file is defined to be a sequence of characters up to and including a special character called the **newline** character represented as `\n`.

Delimiter character is used to separate fields in each line.

The common delimiters are comma, tab, and colon.

```
gcfile = open("graduate course type in Singapore.txt", "r")

for aline in gcfile:
    values = aline.split(",")
    print('In Year ', values[0], ", There are ", values[3], " ", values[1], " graduates in course ", values[2] )

gcfile.close()
```

8.4 Read one line from file

Instead of reading the whole file content, we can read one line from the file.
Below table summarizes methods we can use.
When it reaches end of file, readline() and readlines() will return empty string.

Method Name	Use	Explanation
write	f.write(string)	Add string to the end of the file.
read(n)	f.read()	Reads and returns a string of n characters, or the entire file as a single string if n is not provided.
readline(n)	f.readline()	Returns the next line of the file with all text up to and including the newline character. If n is provided as a parameter than only n characters will be returned if the line is longer than n.
readlines(n)	f.readlines()	Returns a list of strings, each representing a single line of the file. If n is not provided, then all lines of the file are returned. If n is provided, then n characters are read but n is rounded up so that an entire line is returned.

```
infile = open("graduate course type in Singapore.txt", "r")
line = infile.readline()
while line:
    values = line.split(",")
    print('In Year ', values[0], ", There are ",values[3], " ", values[1], " graduates in course ", values[2] )
    line = infile.readline()
infile.close()
```

Priming read

When reach end of file, it treats an empty string as False

This statement reassign variable to next line of the file.

8.5 Write to files

Writing files is like reading files, but we use 'w' or 'wb' mode to write text or binary files, respectively. Remember to close the file, otherwise the last part of the data may be lost.

```
f = open('data.txt', 'w')
f.write('01 02 03')
f.close()
```

There is a **with** statement in Python which can help us **close the opened files automatically**:

```
with open('data.txt', 'w') as f:
    f.write('1234567')
```

8.6 File Input/Output – code challenge

Q: A sample file called `studentdata.txt` contains one line for each student in class. Student name is the first thing on each line, followed by exam scores. The number of scores might be different for each student.

Data file: **studentdata.txt**

```
aaron 10 15 20 30 40  
bill 23 16 19 22  
susan 8 22 17 14 32 17 24 21 2 9 11 17  
jason 12 28 21 45 26 10  
john 14 32 25 16 89
```

Write a program that prints out the names of students that have more than six quiz scores.

8.7 Work with JSON file

JSON is one of the most popular formats for transferring data through APIs nowadays. Python comes with a built-in module for encoding and decoding JSON data.

```
In [14]: import json
```

The following API returns random quote with some extra information in JSON format.

```
In [15]: import requests
r = requests.get('https://api.github.com/')
print(r.text)
```

```
{
  "current_user_url": "https://api.github.com/user",
  "current_user_authorizations_html_url": "https://github.com/settings/connections/applications{/client_id}",
  "authorizations_url": "https://api.github.com/authorizations",
  "code_search_url": "https://api.github.com/search/code?q={query}{&page,per_page,sort,order}",
  "commit_search_url": "https://api.github.com/search/commits?q={query}{&page,per_page,sort,order}",
  "emails_url": "https://api.github.com/user/emails",
  "emojis_url": "https://api.github.com/emojis",
  "events_url": "https://api.github.com/events",
  "feeds_url": "https://api.github.com/feeds",
  "followers_url": "https://api.github.com/user/followers",
  "following_url": "https://api.github.com/user/following{/target}",
  "gists_url": "https://api.github.com/gists{/gist_id}",
  "hub_url": "https://api.github.com/hub",
  "issue_search_url": "https://api.github.com/search/issues?q={query}{&page,per_page,sort,order}",
}
```

By passing the URL to **requests.get** function, we can get the response from the URL, which is usually HTML data.

```
In [16]: json.loads(r.text)
```

```
json.dumps({'key': 'abc', 'value': 'Hello World', 'valid': True})
'{"key": "abc", "value": "Hello World", "valid": true}'
```

We can convert JSON object (saved as a Python string) to a Python dictionary using **json.loads**. Reversely, we can convert a Python dictionary to JSON format string using **json.dumps**