

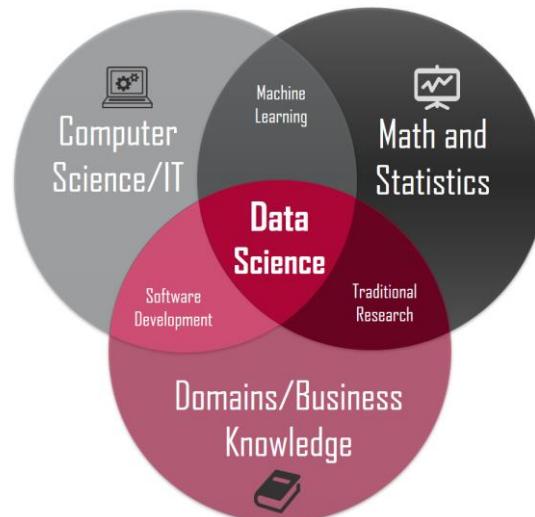
Introduction to Data Science

Lecturer: Ben Galili



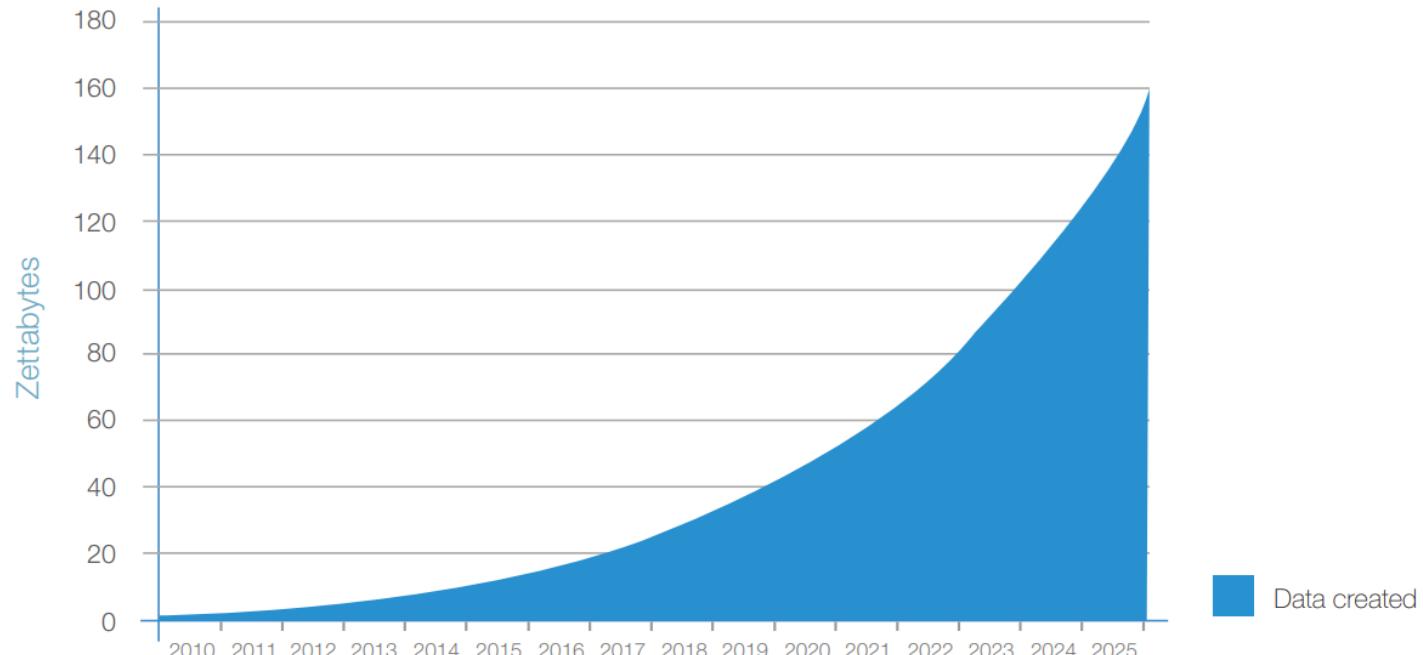
What is Data Science?

- ▶ Data science is the discipline of the extraction of knowledge from data
- ▶ Data science relies on:
 - ▶ Computer science (for data structures, algorithms, visualization, big data support, and general programming)
 - ▶ Statistics (for regressions and inference)
 - ▶ Domain knowledge (for asking questions and interpreting results)



Data Growth

- ▶ Data is eating the world: 163 Zettabytes will be created in 2025



Source: IDC's Data Age 2025 study, sponsored by Seagate, April 2017

Data Science Applications

- ▶ Identifying people based on pictures or voice recordings
- ▶ Finding place names or persons in text
- ▶ Identifying tumors and diseases
- ▶ Predicting the amount of money a person will spend on product X
- ▶ Predicting your company's yearly revenue
- ▶ Proactively identifying car parts that are likely to fail
- ▶ Predicting which team will win the Champions League in soccer
- ▶ Finding oil fields, gold mines, or archeological sites based on existing sites

Facets of Data

- ▶ In data science you'll come across many different types of data, and each of them tends to require different tools and techniques
- ▶ The main categories of data are:
 - ▶ Structured
 - ▶ Unstructured
 - ▶ Natural language
 - ▶ Machine-generated
 - ▶ Graph-based
 - ▶ Audio, video, and images
 - ▶ Streaming

Structured Data

- ▶ Structured data refers to any data that resides in a fixed field within a record or file
- ▶ This includes data contained in relational databases and spreadsheets
- ▶ Has the advantage of being easily entered, stored, queried and analyzed

PowerBI_Test_Data.xlsx - Excel

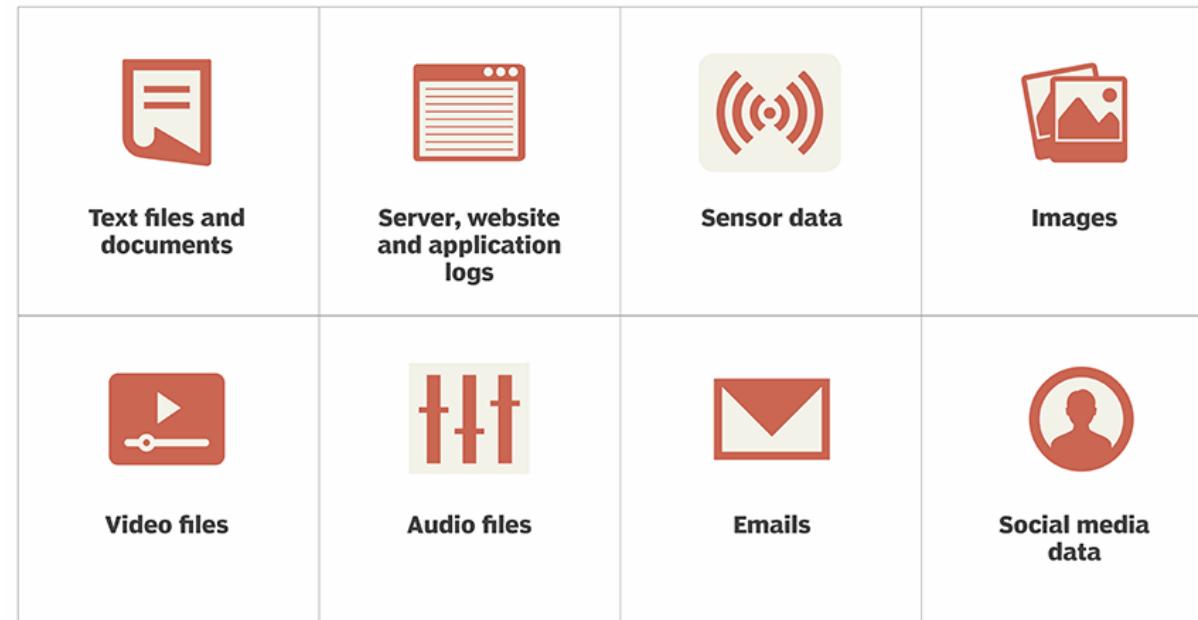
Mark Kaelin

L14

	A	B	C	D	E	F	G	H	I	J
1	Stock Name	Symbol	Shares	Purchase Price	Cost Basis	Current Price	Market Value	Gain/Loss	Dividend/share	Annual Yield
2	Apple	AAPL	100	\$90.00	\$9,000.00	\$144.13	\$14,413.27	\$14,269.14	\$2.28	1.58%
3	Microsoft	MSFT	200	\$32.00	\$6,400.00	\$65.57	\$13,114.14	\$13,048.57	\$1.56	2.38%
4	Salesforce	CRM	150	\$25.00	\$3,750.00	\$82.57	\$12,385.50	\$12,302.93	\$0.00	0.00%
5	Oracle	ORCL	250	\$50.00	\$12,500.00	\$44.56	\$11,138.75	\$11,094.20	\$0.64	1.44%
6	Hewlett Packard Enterprise	HPE	500	\$18.00	\$9,000.00	\$17.69	\$8,842.50	\$8,824.82	\$0.26	1.47%
7	Alphabet	GOOG	100	\$225.00	\$22,500.00	\$833.36	\$83,336.00	\$82,502.64	\$0.00	0.00%
8	Intel	INTC	200	\$22.00	\$4,400.00	\$36.07	\$7,213.00	\$7,176.94	\$1.09	3.02%
9	Cisco	CSCO	225	\$18.00	\$4,050.00	\$33.24	\$7,478.78	\$7,445.54	\$1.16	3.49%
10	Qualcomm	QCOM	185	\$65.00	\$12,025.00	\$56.48	\$10,447.88	\$10,391.40	\$2.12	3.75%
11	Amazon	AMZN	50	\$800.00	\$40,000.00	\$897.64	\$44,882.00	\$43,984.36	\$0.00	0.00%
12	Redhat	RHT	100	\$95.00	\$9,500.00	\$86.26	\$8,626.00	\$8,539.74	\$0.00	0.00%
13	Facebook	FB	1000	\$17.00	\$17,000.00	\$141.64	\$141,640.00	\$141,498.36	\$0.00	0.00%
14	Twitter	TWTR	500	\$45.00	\$22,500.00	\$14.61	\$7,302.55	\$7,287.94	\$0.00	0.00%
15										

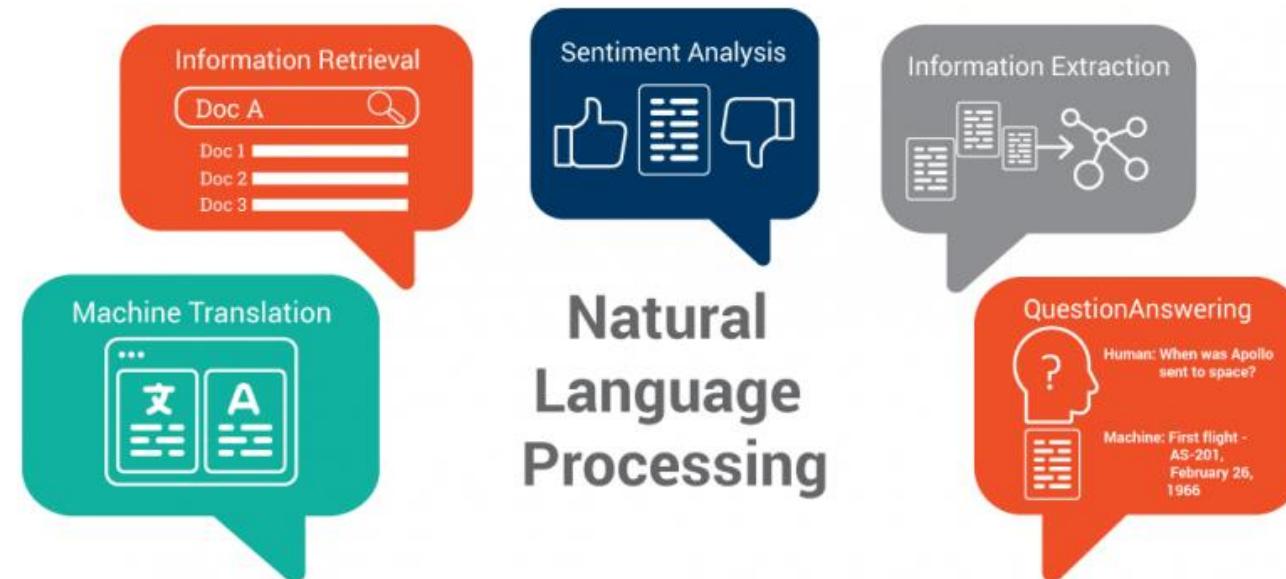
Unstructured Data

- ▶ Data that doesn't have a pre-defined model or isn't organized in a pre-defined way
- ▶ It is estimated that unstructured data account for more than 70%–80% of all data in organizations



Natural Language

- ▶ Natural language is a special type of unstructured data
 - ▶ It's even more challenging to process, because it requires knowledge of specific data science techniques and linguistics
- ▶ Challenges in natural language processing (NLP) frequently involve speech recognition, natural language understanding, and natural language generation



Machine-Generated Data

- ▶ Machine-generated data is information that's automatically created by a computer, process, application, or other machine without human intervention
- ▶ Examples of machine data are web server logs, call detail records, network event logs, and telemetry
- ▶ Machine-generated data is characterized by high volume and speed

```

157.55.39.229 - 33years [06/Jul/2014:00:01:57 +0000] "GET /assets/js/main.js?v2014.03.03a HTTP/1.1" 200 6110 "-" "Mozilla/5.
157.55.39.229 - 33years [06/Jul/2014:00:01:57 +0000] "GET /assets/js/main.js?v2014.03.03a HTTP/1.1" 200 6110 "-" "Mozilla/5.
207.46.13.101 - 33years [06/Jul/2014:00:02:03 +0000] "GET /products/workbench/design-run-jobs HTTP/1.1" 200 78102 "-" "Mozil
199.58.86.206 - 33years [06/Jul/2014:00:02:10 +0000] "GET /robots.txt HTTP/1.0" 301 237 "-" "Mozilla/5.0 (compatible; MJ12bc
199.58.86.206 - 33years [06/Jul/2014:00:02:11 +0000] "GET /robots.txt HTTP/1.0" 200 79 "-" "Mozilla/5.0 (compatible; MJ12bot
199.58.86.206 - 33years [06/Jul/2014:00:02:12 +0000] "GET / HTTP/1.0" 301 227 "-" "Mozilla/5.0 (compatible; MJ12bot/v1.4.5;
199.58.86.206 - 33years [06/Jul/2014:00:02:13 +0000] "GET /robots.txt HTTP/1.0" 200 79 "-" "Mozilla/5.0 (compatible; MJ12bot
199.58.86.206 - 33years [06/Jul/2014:00:02:14 +0000] "GET / HTTP/1.0" 200 71409 "-" "Mozilla/5.0 (compatible; MJ12bot/v1.4.5
207.46.13.108 - 33years [06/Jul/2014:00:02:42 +0000] "GET /solutions/data-masking/masking HTTP/1.1" 302 - "-" "Mozilla/5.0 (
207.46.13.108 - 33years [06/Jul/2014:00:02:42 +0000] "GET /products/workbench HTTP/1.1" 200 78784 "-" "Mozilla/5.0 (compatib
211.244.83.24 - 33years [06/Jul/2014:00:03:11 +0000] "GET /robots.txt HTTP/1.1" 301 237 "http://search.daum.net/" "Mozilla/5
21.244.83.248 - 33years [06/Jul/2014:00:03:12 +0000] "GET /robots.txt HTTP/1.1" 200 79 "http://search.daum.net/" "Mozilla/5.
21.244.83.248 - 33years [06/Jul/2014:00:03:13 +0000] "GET / HTTP/1.1" 301 227 "http://search.daum.net/" "Mozilla/5.0 (compat
21.244.83.248 - 33years [06/Jul/2014:00:03:14 +0000] "GET / HTTP/1.1" 200 71409 "http://search.daum.net/" "Mozilla/5.0 (comp
94.228.34.211 - 33years [06/Jul/2014:00:04:04 +0000] "GET /clientarea/forum/feed/_ HTTP/1.1" 302 210 "-" "magpie-crawler/1.1
94.228.34.211 - 33years [06/Jul/2014:00:04:04 +0000] "GET /support HTTP/1.1" 200 49529 "-" "magpie-crawler/1.1 (U; Linux amc
180.76.150.57 - 33years [06/Jul/2014:00:04:07 +0000] "GET /solutions/test-data HTTP/1.1" 200 95155 "-" "Mozilla/5.0 (compati
66.228.61.183 - 33years [06/Jul/2014:00:05:04 +0000] "GET / HTTP/1.1" 200 129773 "-" "-"
66.228.61.183 - 33years [06/Jul/2014:00:05:04 +0000] "GET /products HTTP/1.1" 200 142021 "-" -
66.228.61.183 - 33years [06/Jul/2014:00:05:04 +0000] "GET /blog/_ HTTP/1.1" 200 3833 "-" -
207.46.13.101 - 33years [06/Jul/2014:00:05:14 +0000] "GET /customers/industries/telco-cable HTTP/1.1" 200 55063 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36"
66.249.64.209 - 33years [06/Jul/2014:00:05:55 +0000] "GET /blog/category/data-transformation2/_ HTTP/1.1" 200 91669 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36"
157.55.39.310 - 33years [06/Jul/2014:00:06:00 +0000] "GET /company/about-iri-the-cosort-company/recognition HTTP/1.1" 200 56
66.249.64.209 - 33years [06/Jul/2014:00:06:04 +0000] "GET /blog/data-protection/data-masking-and-data-encryption-are-not-the-same HTTP/1.1" 200 78176 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36"
157.55.39.229 - 33years [06/Jul/2014:00:06:27 +0000] "GET /products/fieldshield/platforms-pricing HTTP/1.1" 200 78176 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36"
207.133.71.34 - 33years [06/Jul/2014:00:06:36 +0000] "GET /blog/wp-content/themes/thesis_182/custom/custom.css HTTP/1.1" 200 95687 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36"
180.76.50.162 - 33years [06/Jul/2014:00:06:38 +0000] "GET /products/workbench/data-sources HTTP/1.1" 302 234 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36"
180.76.60.316 - 33years [06/Jul/2014:00:06:38 +0000] "GET /products/workbench/data-sources HTTP/1.1" 200 80112 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36"
207.46.13.108 - 33years [06/Jul/2014:00:06:41 +0000] "GET /solutions/sort-replacements HTTP/1.1" 200 95687 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36"

```

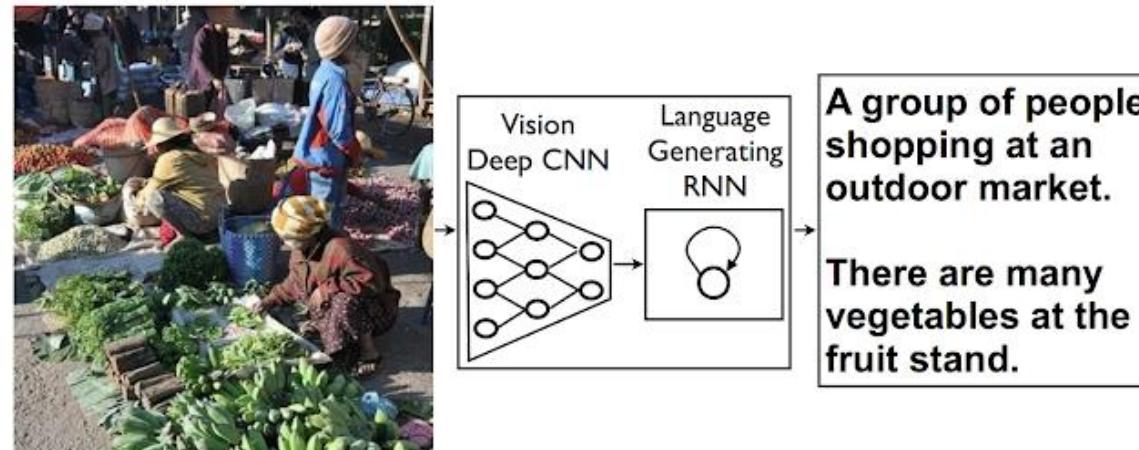
Graph-Based Data

- ▶ Graph data is data that focuses on the relationship or adjacency of objects
- ▶ Graph-based data is a natural way to represent social networks
- ▶ Its structure allows you to calculate various metrics such as the influence of a person and the shortest path between two people



Audio, Image and Video

- ▶ Audio, image, and video are data types that pose specific challenges to a data scientist
- ▶ Tasks that are trivial for humans, such as recognizing objects in pictures, turn out to be challenging for computers
- ▶ Deep Learning techniques have been applied to many image processing and computer vision problems with great success in recent years



Streaming Data

- ▶ Streaming data is data that is generated continuously by many different data sources, which typically send in the data records simultaneously, and in small sizes
- ▶ Examples include live sporting or music events, the stock market, and “What’s trending” on Twitter
- ▶ Such data should be processed incrementally using stream processing techniques without having access to all of the data

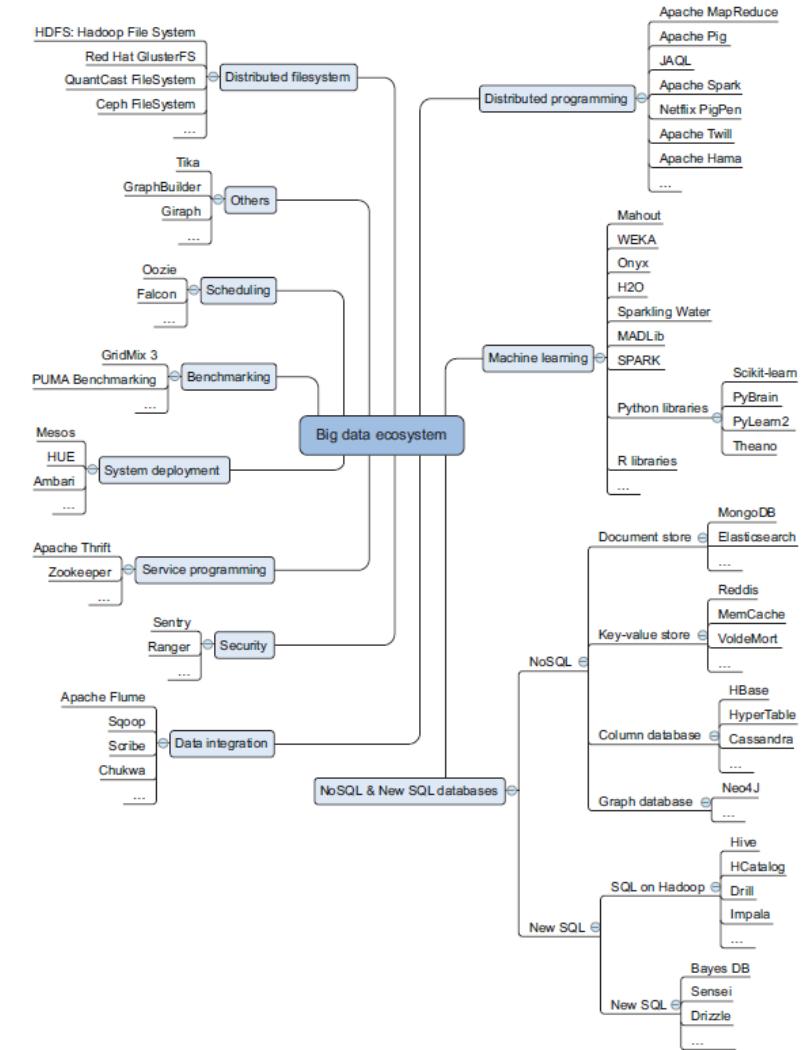


- ▶ Big data is data sets that are so large or complex that traditional data techniques are inadequate to deal with them (such as relational databases)
 - ▶ The characteristics of big data are often referred to as the four Vs:
 - ▶ Volume – How much data is there?
 - ▶ Variety – How diverse are different types of data?
 - ▶ Velocity – At what speed is new data generated?
 - ▶ Veracity – How accurate is the data?



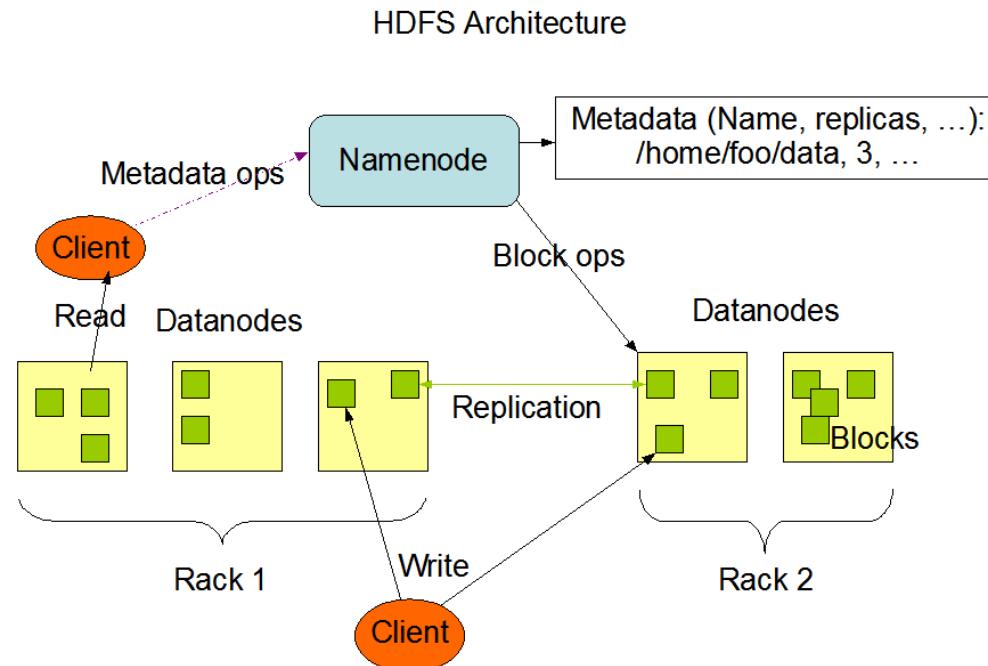
The Big Data Ecosystem

- ▶ The big data landscape consists of many types of technologies
- ▶ Data scientists focus mainly on:
 - ▶ Distributed file systems
 - ▶ Distributed programming
 - ▶ NoSQL Databases
 - ▶ Machine learning



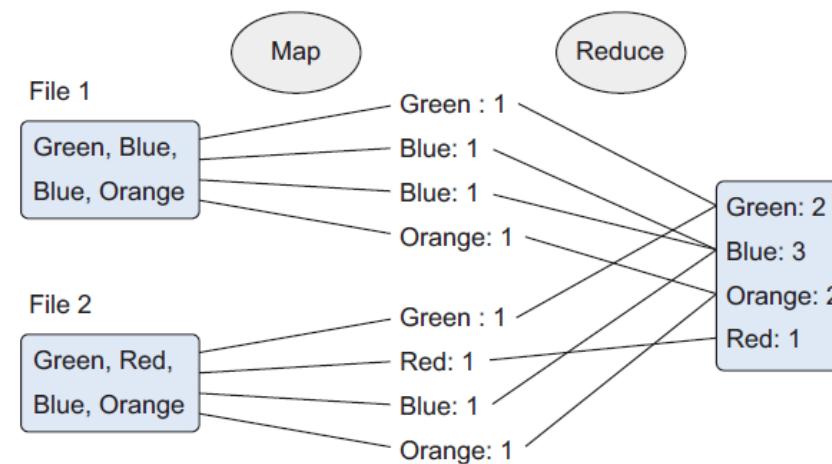
Distributed File Systems

- ▶ A distributed file system is similar to a normal file system, except that it runs on multiple servers at once
- ▶ The best-known distributed file system is the **Hadoop File System (HDFS)**
 - ▶ It is an open source implementation of the Google File System



Distributed Programming Frameworks

- ▶ In **distributed computing**, a problem is divided into many tasks, each of which is solved by a single computer
 - ▶ The computers communicate with each other via message passing
- ▶ **MapReduce** is a software framework for easily writing applications which process vast amounts of data in-parallel on large clusters



NoSQL Databases

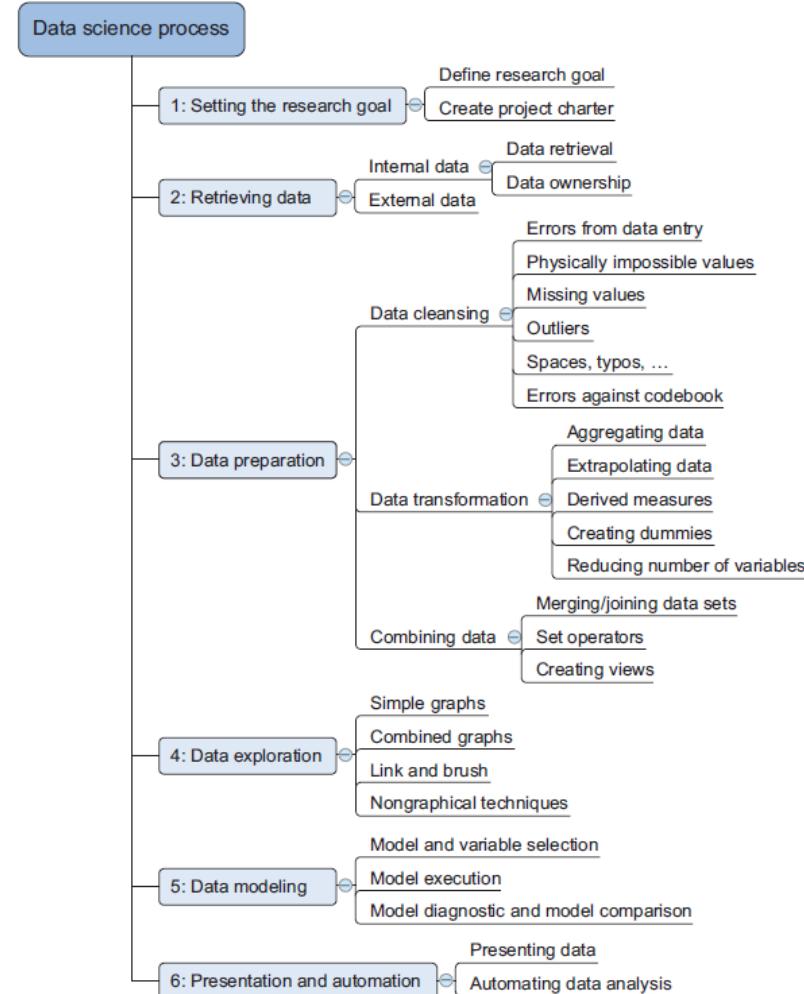
- ▶ Traditional relational databases have shortcomings that don't allow them to scale well, and they have no way to handle streaming, graph, or unstructured data
- ▶ **NoSQL** (Not-Only SQL) databases allow for a virtually endless growth of data
- ▶ Many different types of NoSQL databases exist, but they can be categorized into:

Document Database	Graph Databases
 Couchbase  mongoDB	 Neo4j  InfiniteGraph The Distributed Graph Database
Wide Column Stores	Key-Value Databases
 AEROSPIKE  riak	 HYPERTABLE INC.  Apache HBASE Amazon SimpleDB

@cloudtxt <http://www.aryannava.com>

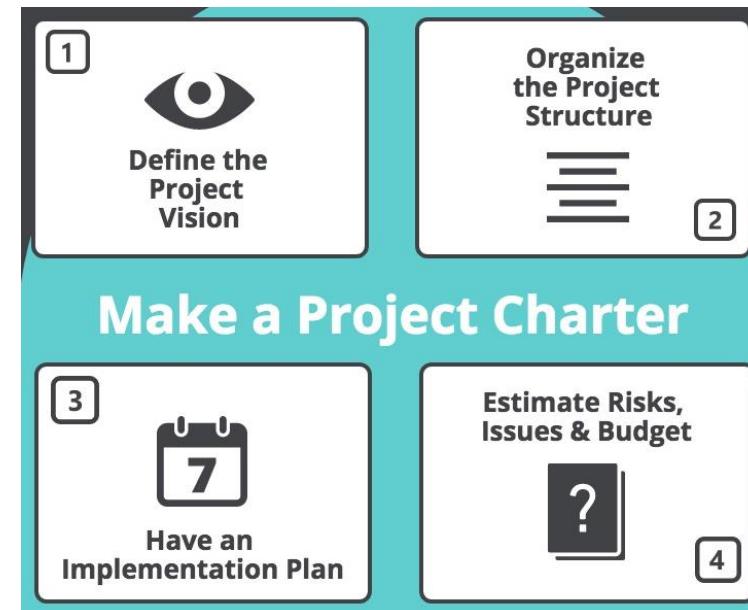
The Data Science Process

- ▶ The typical data science process consists of six steps shown on the right figure
- ▶ In reality you won't progress in a linear way from step 1 to step 6
- ▶ Often you'll regress and iterate between the different phases



Defining the Research Goals

- ▶ A project starts by understanding the *what, why, and how* of your project
- ▶ The outcome should be a clear research goal, a good understanding of the context, well-defined deliverables, and a plan of action with a timetable
- ▶ This information is then best placed in a **project charter**



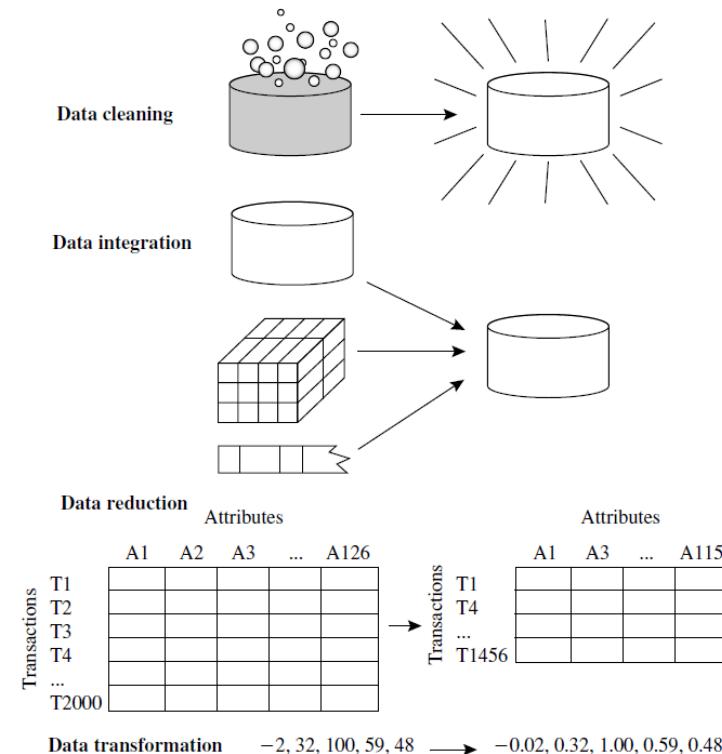
Retrieving Data

- ▶ The first steps of your process focus on getting high-quality data
- ▶ Finding data even within your own company can sometimes be a challenge
 - ▶ As companies grow, their data becomes scattered around many places
- ▶ Getting access to data is another difficult task
 - ▶ Organizations often have policies in place so everyone has access to only what they need



Data Preparation

- ▶ Raw data is like “a diamond in the rough”: it needs polishing to be of any use to you
- ▶ In data science there’s a well-known saying: *Garbage in equals garbage out*
- ▶ Data preparation involves the following steps to get the raw data ready for analysis:



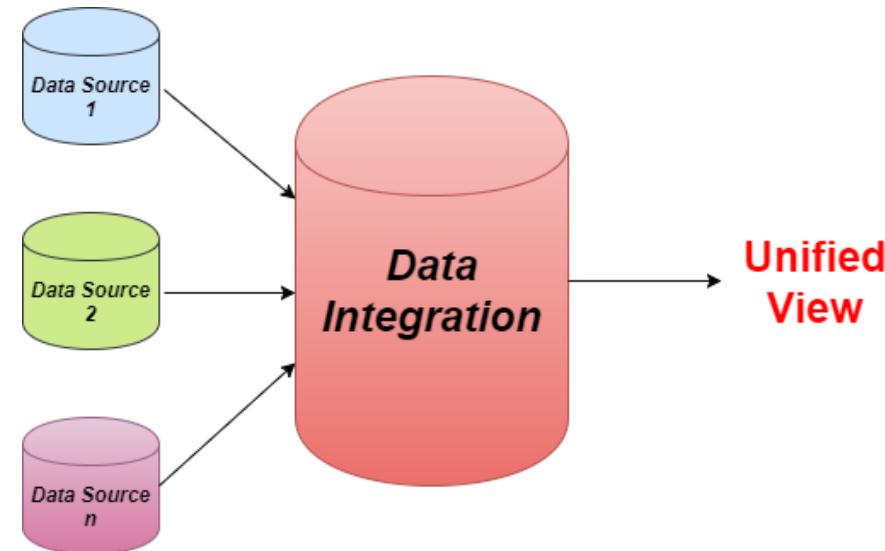
Data Cleaning

- ▶ Data cleaning focuses on removing errors in your data so your data becomes a true and consistent representation of the processes it originates from
- ▶ Common data errors:

Error description	Possible solution
<i>Errors pointing to false values within one data set</i>	
Mistakes during data entry	Manual overrules
Redundant white space	Use string functions
Impossible values	Manual overrules
Missing values	Remove observation or value
Outliers	Validate and, if erroneous, treat as missing value (remove or insert)
<i>Errors pointing to inconsistencies between data sets</i>	
Deviations from a code book	Match on keys or else use manual overrules
Different units of measurement	Recalculate
Different levels of aggregation	Bring to same level of measurement by aggregation or extrapolation

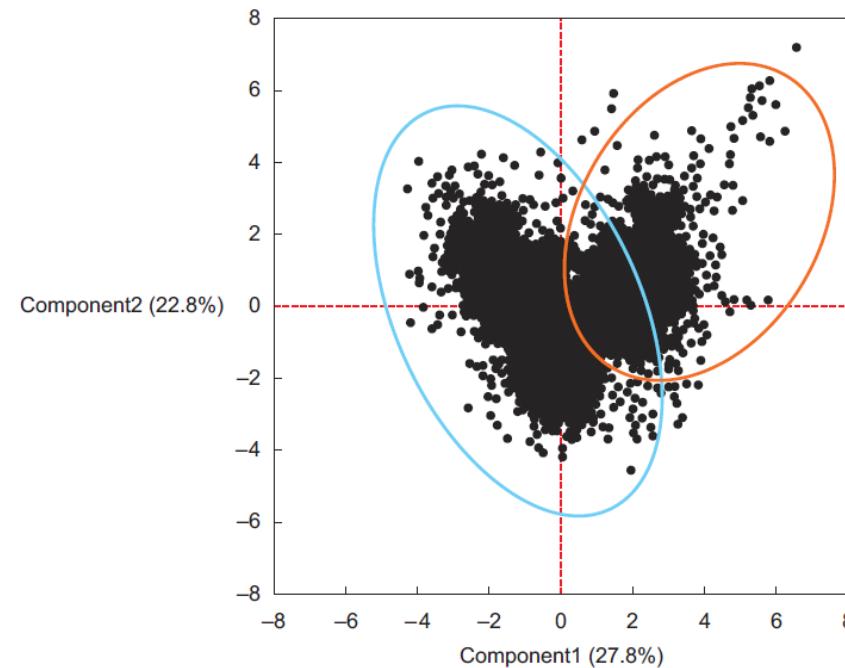
Data Integration

- ▶ Data varies in size, type, and structure, ranging from databases and Excel files to text documents
- ▶ Data integration combines data from multiple sources to form a coherent data store



Feature Selection

- ▶ Often you have too many features (variables) in the model, which makes the model difficult to handle
- ▶ Data scientists use methods such as PCA (Principal Component Analysis) to reduce the number of variables but retain the maximum amount of data



Data Transformation

- ▶ Data transformation convert the data into appropriate forms for modeling
- ▶ For example, in **normalization**, attribute data are scaled so as to fall within a small range such as 0.0 to 1.0
- ▶ **Data discretization** transforms numeric data by mapping values to interval or concept labels

Age	10,11,13,14,17,19,30, 31, 32, 38, 40, 42,70 , 72, 73, 75
-----	--

Table: Before discretization

Age	10,11,13,14,17,19,	30, 31, 32, 38, 40, 42,	70 , 72, 73, 75
	Young	Mature	Old

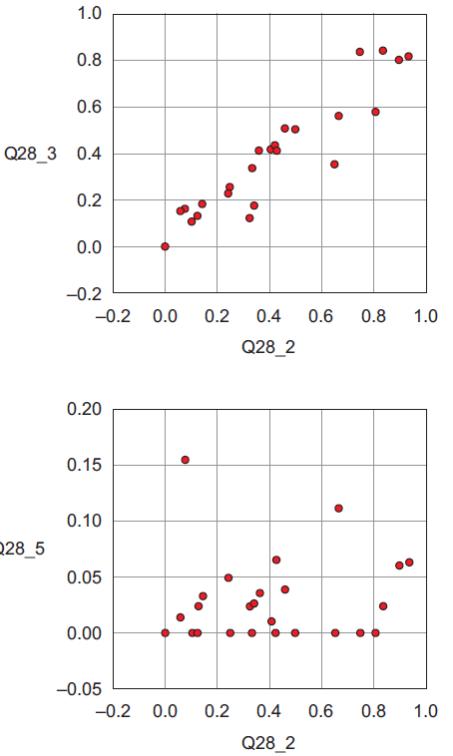
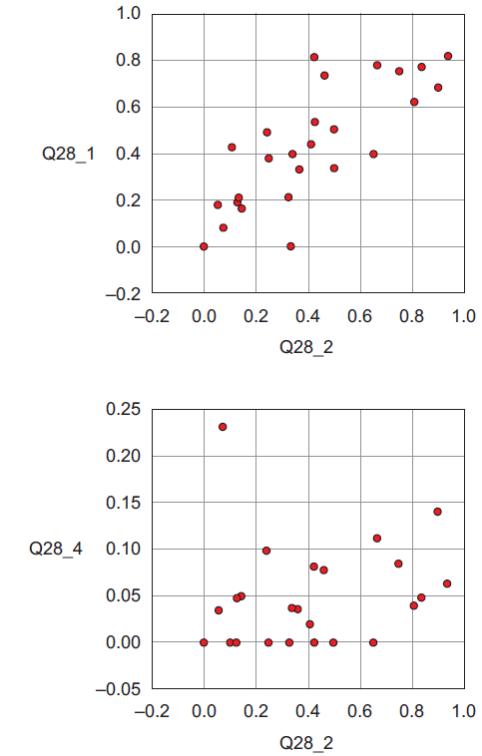
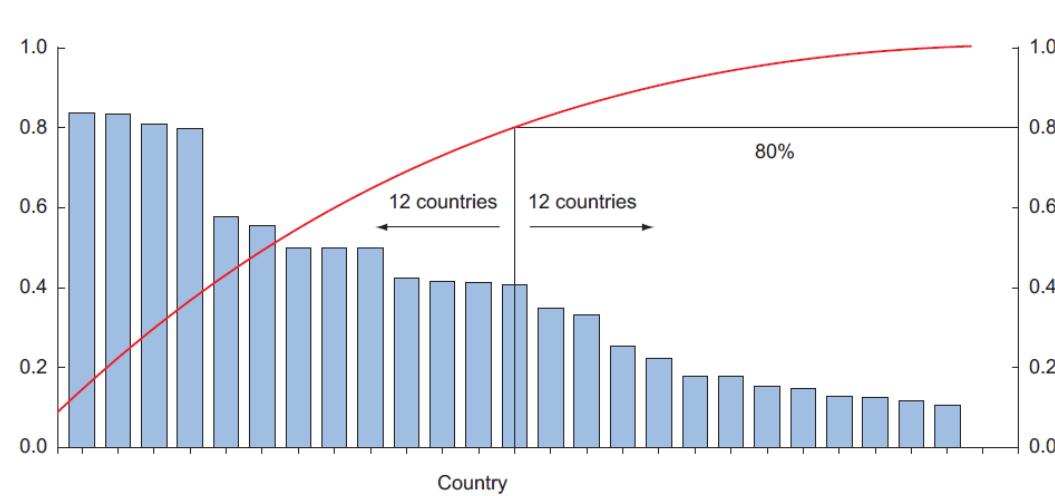
Table: After discretization

Exploratory Data Analysis

- ▶ During this phase you take a deep dive into the data
- ▶ You'll look for patterns, correlations, and deviations based on visual and descriptive techniques
- ▶ The insights you gain from this phase will enable you to start modeling
- ▶ It's common that you'll still discover anomalies you missed before, forcing you to take a step back and fix them

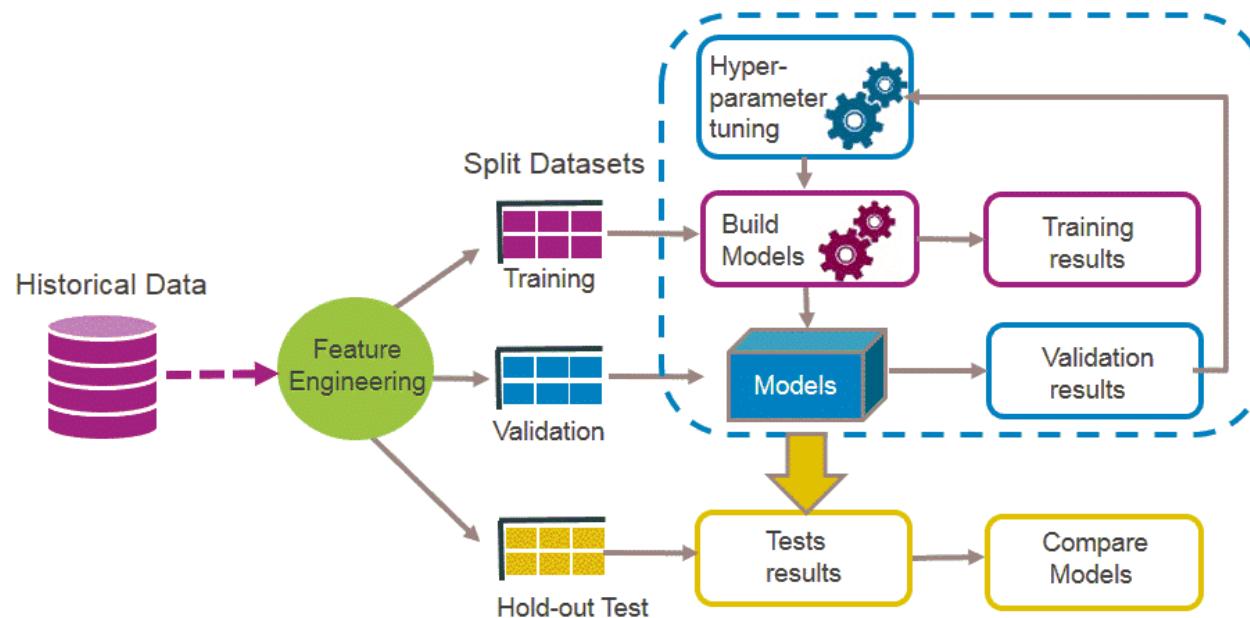
Visualization

- The visualization techniques you use in this phase range from simple line graphs or histograms, to more complex diagrams such as network graphs



Build the Models

- ▶ With clean data in place and good understanding of the content, you're ready to build the model
- ▶ In this stage you'll use techniques from machine learning, data mining, and statistics
- ▶ Most models consist of the following main steps:



Presenting the Results

- ▶ After you've successfully analyzed the data and built a well-performing model, you're ready to present your findings to the stakeholders
- ▶ This is where your *soft skills* will be most useful for convincing the business that your findings will indeed change the business process as expected
- ▶ Sometimes you'll need to automate your analysis process for repetitive reuse and integration with other tools



Issues and Challenges in Data Science

- ▶ Large data
 - ▶ Number of variables (features/dimensions), number of instances (cases)
 - ▶ Multi gigabyte, terabyte databases
 - ▶ Efficient algorithms, parallel processing
- ▶ Missing and noisy data
- ▶ Changing data
- ▶ High dimensionality
 - ▶ Potential for spurious patterns
 - ▶ Dimensionality reduction
- ▶ Overfitting
 - ▶ Models adapt to the noise in training data, instead of finding the general patterns
- ▶ Use of domain knowledge
- ▶ Understandability of patterns

Introduction to Python

Lecturer: Ben Galili

What is Python?

- ▶ Python is a powerful high-level, interpreted, object-oriented programming language
- ▶ Created by Guido van Rossum and first released in 1991
- ▶ It has a simple and easy-to-use syntax
- ▶ Features a dynamic type system and automatic memory management
- ▶ Has a rich variety of native data structures such as lists, tuples, sets and dictionaries
- ▶ Allows you to create wide range of applications, from Web applications and scientific applications to desktop graphical user interfaces



What is Python?

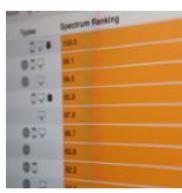
- ▶ Python is ranked as the leading programming language in all recent surveys

31 Jul 2018 | 15:00 GMT

The 2018 Top Programming Languages

Python extends its lead, and Assembly enters the Top Ten

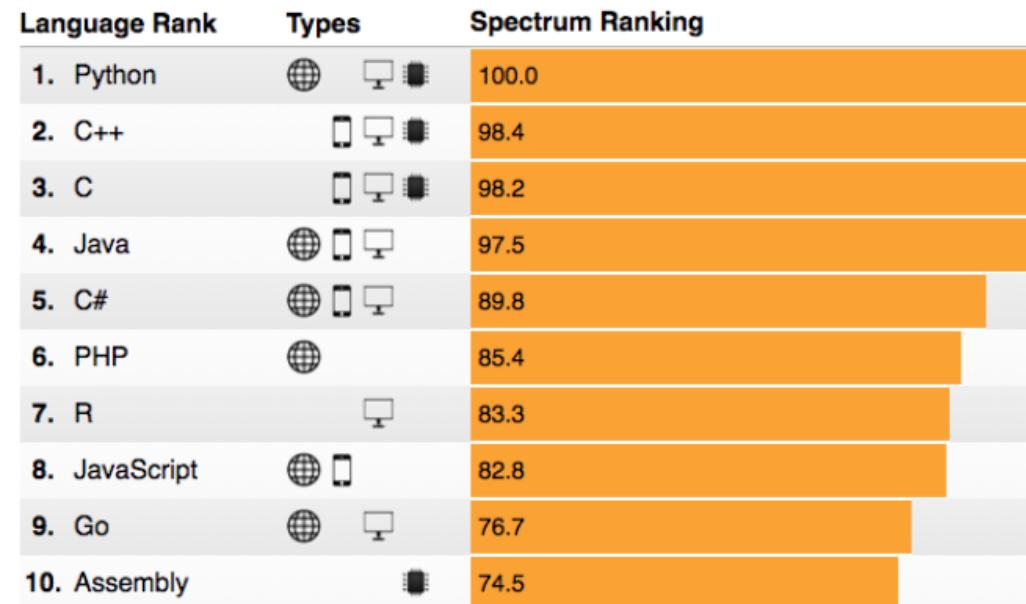
By Stephen Cass



Explore the Interactive Rankings

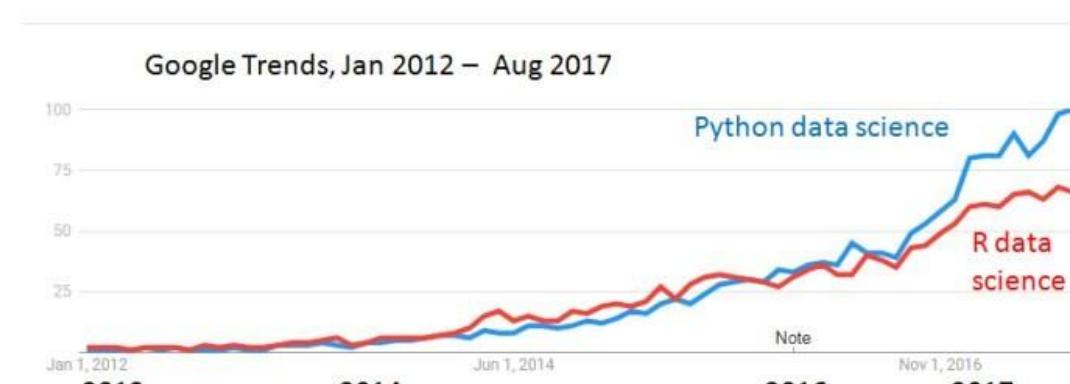
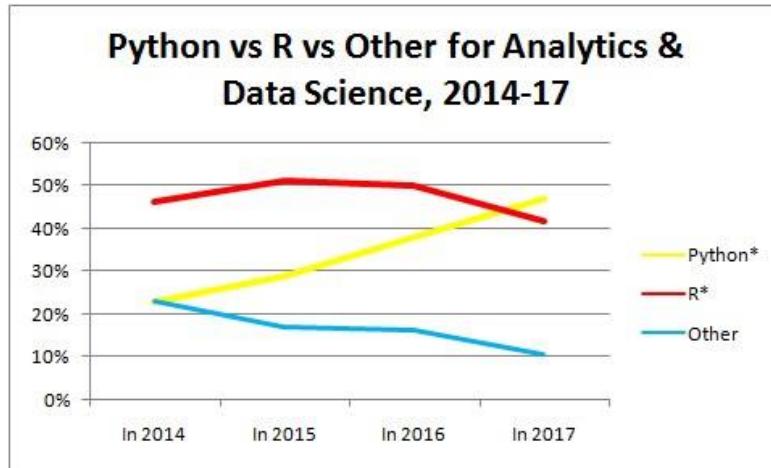
Welcome to *IEEE Spectrum's* [fifth annual interactive ranking of the top programming languages](#). Because no one can peer over the shoulders of every coder out there, anyone attempting to measure the popularity of computer languages must rely on proxy measures of relative popularity. In our case, this means combining metrics from multiple sources to rank 47 languages. But recognizing that different programmers have different needs and domains of interest, we've chosen not to blend all those metrics up into One Ranking to Rule Them All.

Instead, our interactive app lets you choose how these metrics are weighted when they are combined, so you can



Python vs. R for Data Science

- ▶ Python and R are the two most popular programming languages used by data analysts and data scientists
- ▶ R is a language specialized for handling statistics and big data
- ▶ The existence of high-quality Python libraries for both statistics and machine learning makes Python a more attractive jumping-off point than the more specialized R



Python Vs. C

- ▶ The following program output a list of names on separate lines
- ▶ The same program is written in C and in Python

```
names = ["Isaac Newton", "Marie Curie", "Paul Dirac"]
for name in names:
    print(name)
```

```
#include <stdio.h>
#include <string.h>
#define MAX_STRING_LENGTH 20
#define NUMBER_OF_STRINGS 3

int main()
{
    char names[NUMBER_OF_STRINGS][MAX_STRING_LENGTH + 1];
    int i;

    strcpy(names[0], "Isaac Newton");
    strcpy(names[1], "Marie Curie");
    strcpy(names[2], "Paul Dirac");

    for (i = 0; i < NUMBER_OF_STRINGS; i++) {
        printf("%s\n", names[i]);
    }

    return 0;
}
```

Python Advantages

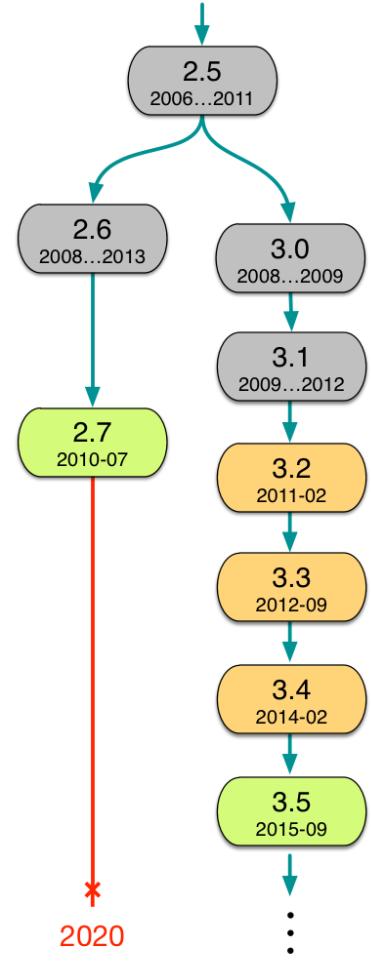
- ▶ Its clean and simple syntax makes writing Python programs fast
 - ▶ Python's syntax aims to ensure that “There should be one – and preferably only one – obvious way to do it”
- ▶ It's free – Python and its associated libraries are free of cost and open source
 - ▶ Unlike commercial offerings such as Matlab or Mathematica
- ▶ Cross-platform support: Python is available for every commonly available operating system, including Windows, Unix, Linux and Mac OS
 - ▶ It is possible to write code that will run on any platform without modification
- ▶ A “multi-paradigm” language that contains the best features from the procedural, object-oriented and functional programming paradigms
- ▶ Has a large library of modules and packages that extend its functionality
 - ▶ Many of these are available as part of the “standard library” provided with Python itself
 - ▶ Others, including NumPy, Pandas and Scikit-learn used in data science can be downloaded separately for no cost

Python Disadvantages

- ▶ The speed of execution of a Python program is not as fast as some other, fully compiled languages such as C and Fortran
 - ▶ For heavily numerical work, the NumPy and SciPy libraries alleviate this to some extent by using compiled-C code “under the hood”, but at the expense of some reduced flexibility
 - ▶ For most application the speed difference is not noticeable
- ▶ It is hard to hide or obfuscate the source code of a Python program to prevent others from copying or modifying it
- ▶ There are some compatibility issues between different Python versions
 - ▶ Especially between Python versions 2 and 3

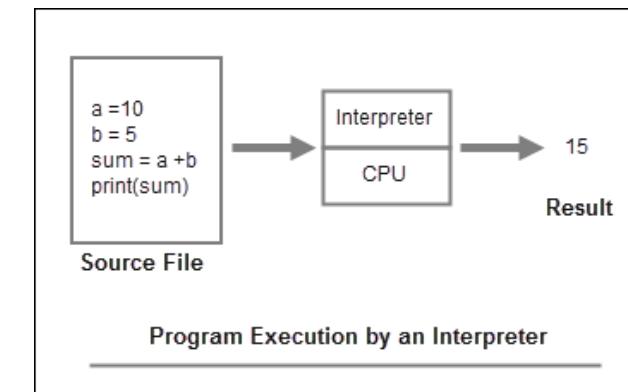
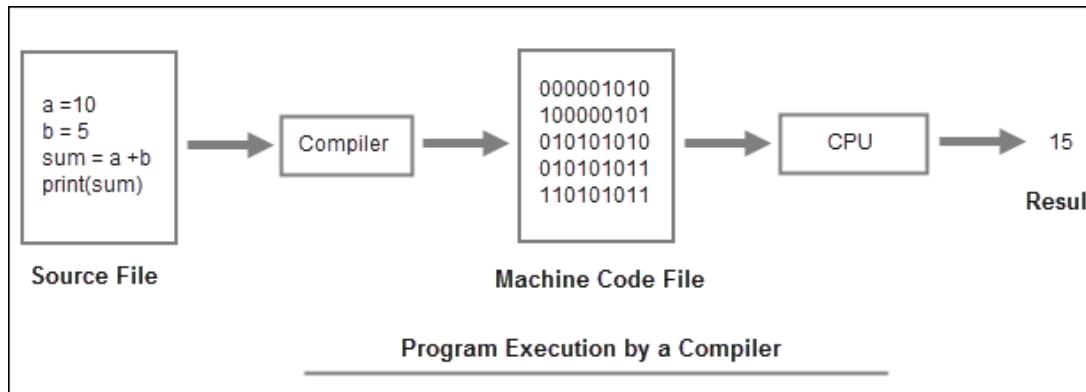
Python Versions

- ▶ Python 2.x is legacy, Python 3.x is the present and future of the language
 - ▶ All recent standard library improvements are only available in Python 3.x
- ▶ Python 3 is not backward-compatible with Python 2
- ▶ Python 3.7.0 is the newest major release of the Python language
 - ▶ Released on June 27, 2018



Compiler vs. Interpreter

- ▶ A program written in a high-level language is called a **source code**
- ▶ We need to convert the source code into machine code (written in binary) before the program can be executed
- ▶ This can be accomplished by compilers and interpreters:
 - ▶ A **compiler** scans the entire program and translates it as a whole into machine code
 - ▶ An **interpreter** translates the program one statement at a time



Compiler vs. Interpreter

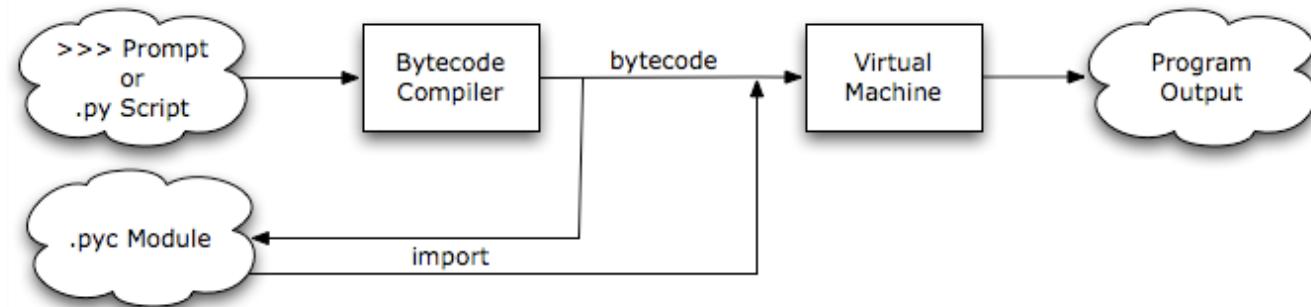
Compiler	Interpreter
Translates the entire program at once	Translates the program line-by-line
Programs are compiled once and run anytime	Programs are interpreted line-by-line every time they are run
Compiled programs execute control statements (like if-else) faster than interpreted programs	Interpreter executes control statements at a slower speed
Compiled programs take more memory because the entire object code needs to reside in memory	Interpreter doesn't generate intermediate object code or machine code, thus is more memory efficient
Errors are reported after the entire program is checked	An error is reported as soon as the first error is encountered, the rest of the program is not checked until the error is fixed
A program is not allowed to run until it is completely error free, which makes it harder to debug	The program runs from the first line and stops execution only if it encounters an error, thus debugging is easier
Examples for programming languages that use compilers: C/C++, Java, C#	Examples for programming languages that use interpreters: Python, PHP, Perl, Ruby, Matlab, Javascript, Lisp

Bytecode and Virtual Machines

- ▶ Python is a byte-code compiled system, in which the source code is translated to an intermediate language known as **bytecode**
- ▶ Bytecode is not a machine code of any particular computer, thus it can run on any CPU architecture without modification, which is a huge benefit over compiled code
- ▶ On the other hand it is lower level than the original source code, thus it runs quicker than interpreted code
- ▶ Bytecode is run on a **Virtual Machine (VM)**, which interprets it into machine-code instructions that are understandable by the specific architecture the program is running on
- ▶ Python source code (.py) can be compiled into different byte codes, such as CPython bytecode (.pyc), IronPython (.Net), or Jython (JVM)

The Python Interpreter

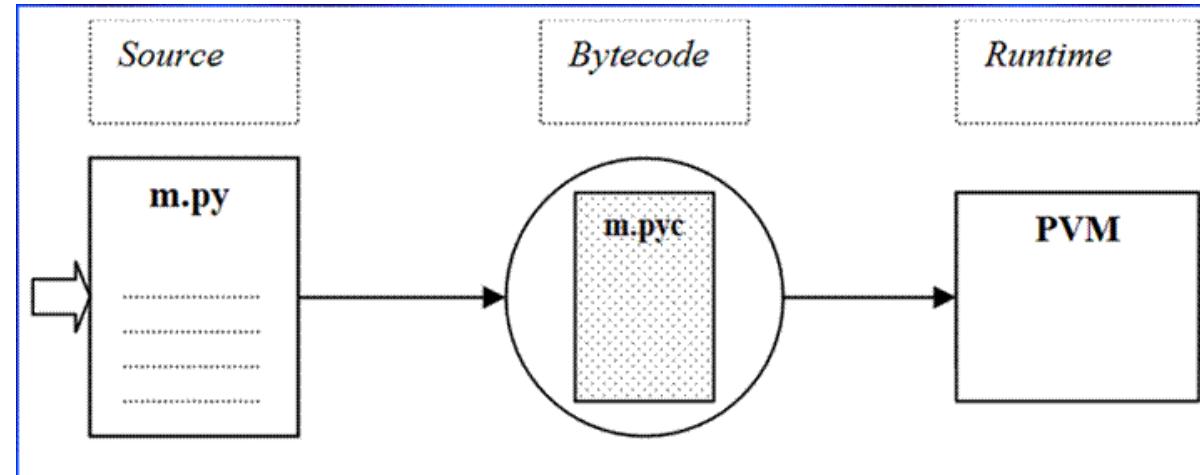
- ▶ The Python interpreter consists of two parts:
 - ▶ A **bytecode compiler** that translates the Python source code into bytecode
 - ▶ A **Python Virtual Machine** (PVM), which executes Python bytecode



- ▶ The Python interpreter can be used in two modes:
 - ▶ Interactive mode – a command line shell which gives immediate feedback for each statement
 - ▶ Script mode – the Python interpreter runs the entire program from the source file

C_Python

- ▶ CPython is the standard and most-widely implementation of the Python programming language, written in C
- ▶ When you run a Python script (.py), the CPython interpreter, caches the translated bytecode in **.pyc** files
- ▶ When you next run your code, the Bytecode file can be used instead of your source code, which will result in quicker execution times



Just-In-Time Compilation

- ▶ A **JIT compiler** compiles parts of the bytecodes into machine code, just before they are about to be executed (hence the name “just-in-time”)
 - ▶ The compiled code is then cached and can be reused later, so the program can run faster
- ▶ **PyPy** is an alternative implementation of the Python programming language which often runs faster than CPython
 - ▶ Because it is a just-in-time compiler, while CPython is an interpreter
- ▶ Most Python code runs well on PyPy, except for code that relies on CPython extensions, which either does not work or incurs some overhead when run in PyPy

Python Installation

- ▶ The official website of Python is <http://www.python.org>
- ▶ It contains full and easy-to-follow instructions for downloading Python
- ▶ However, there are several full distributions which include the data science libraries such as NumPy, SciPy, Scikit-learn save you from having to download and install these yourself
- ▶ For Windows, the Anaconda distribution is typically used

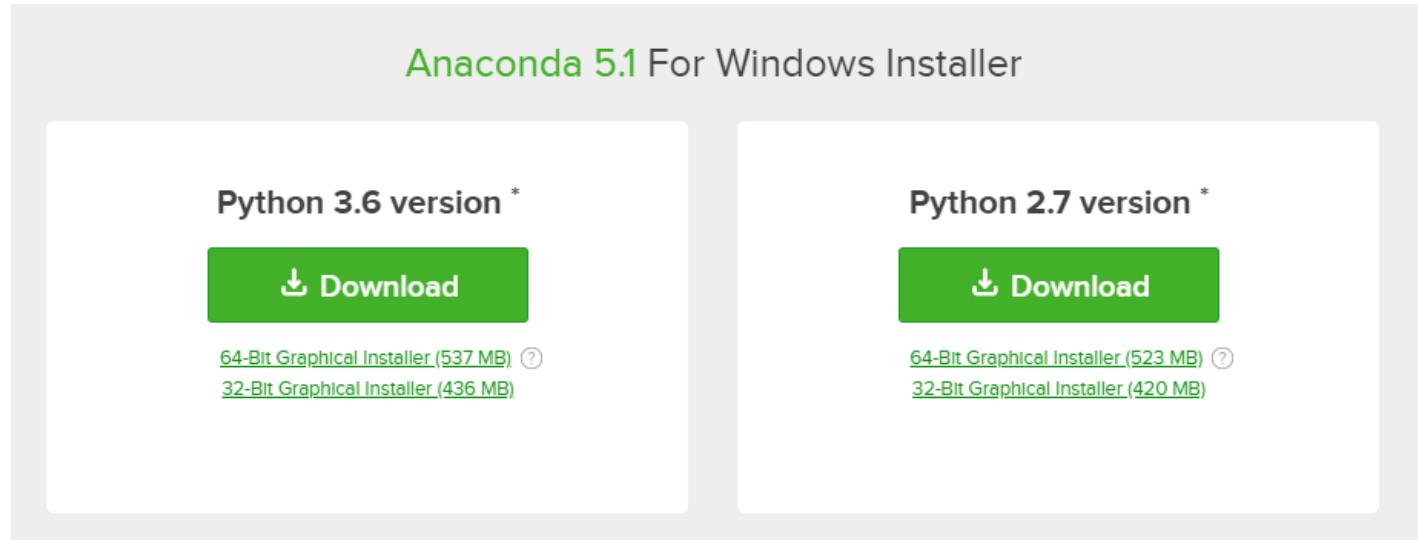
Anaconda Distribution

- ▶ Anaconda is the most popular Python data science platform
- ▶ Anaconda is a distribution of the Python and R programming languages for data science and machine learning related applications
- ▶ It also installs the Jupyter Notebook
- ▶ Includes a collection of over 1,000 open source data science packages
- ▶ Package versions are managed by the package management system conda
- ▶ In contrast to pip, conda can handle library dependencies outside of the Python packages as well as the Python packages themselves



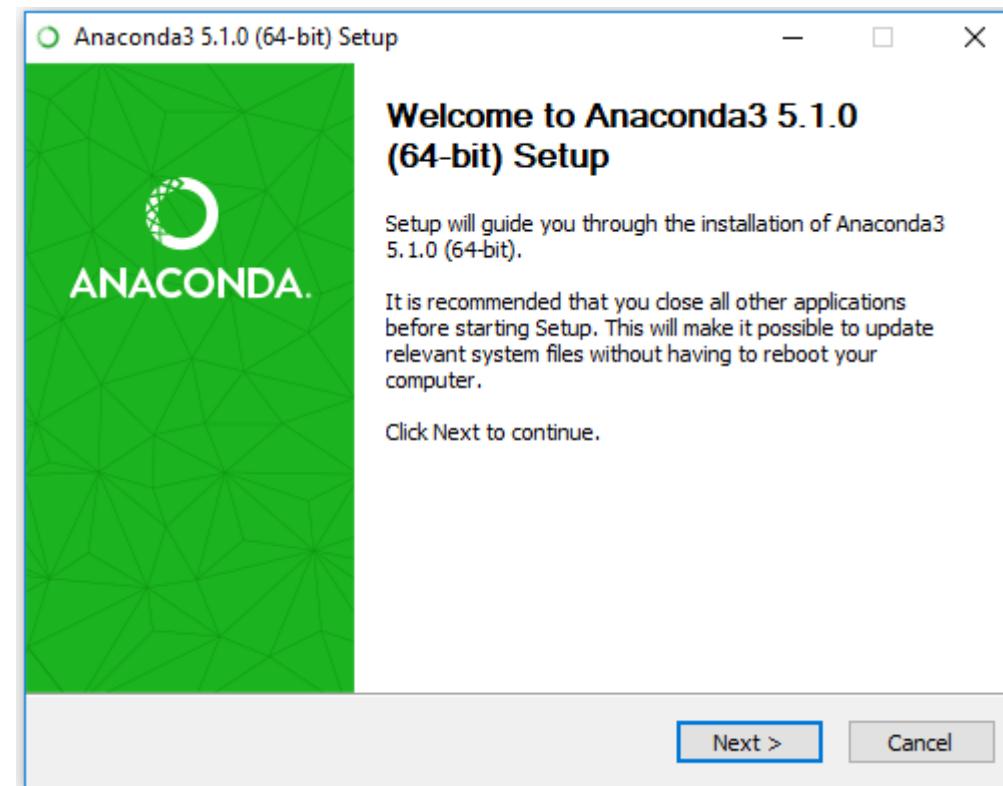
Installing Anaconda

- ▶ Go to <https://www.anaconda.com/download/>
- ▶ Download the Python 3.6 version

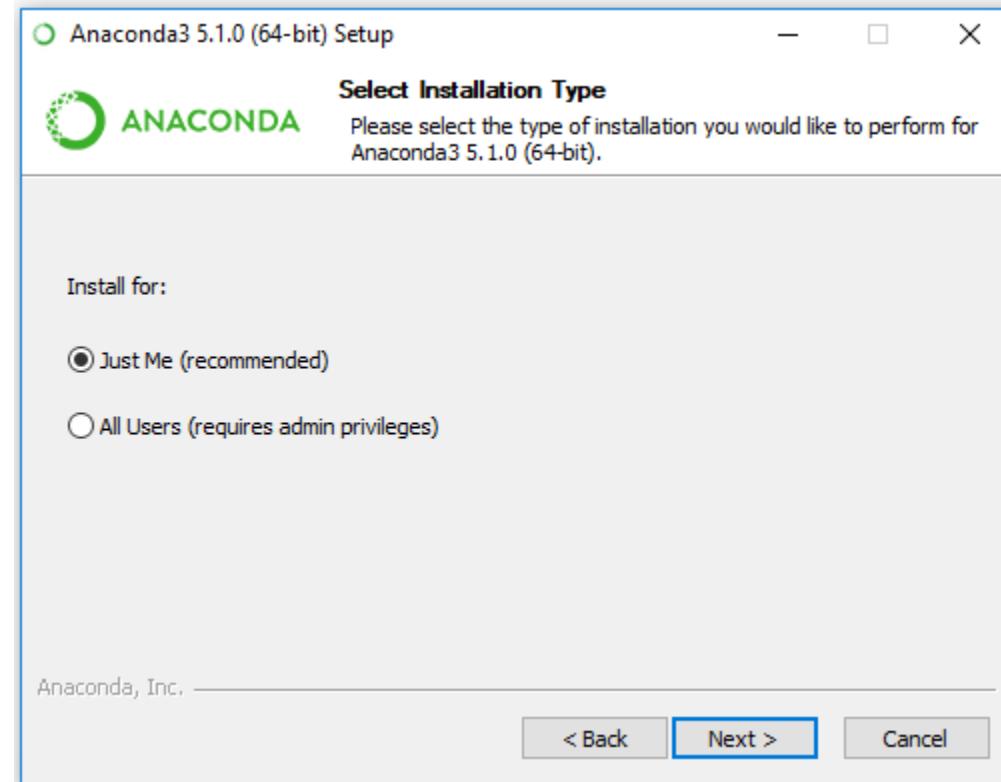


Installing Anaconda

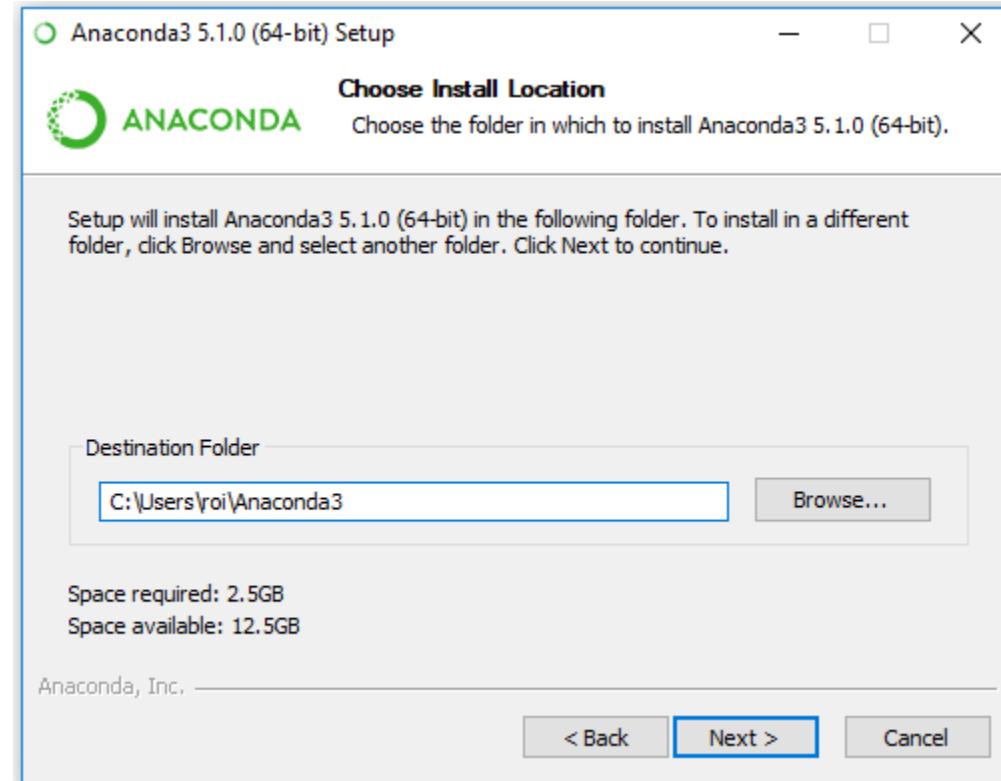
- ▶ Double click the executable file to start the installation



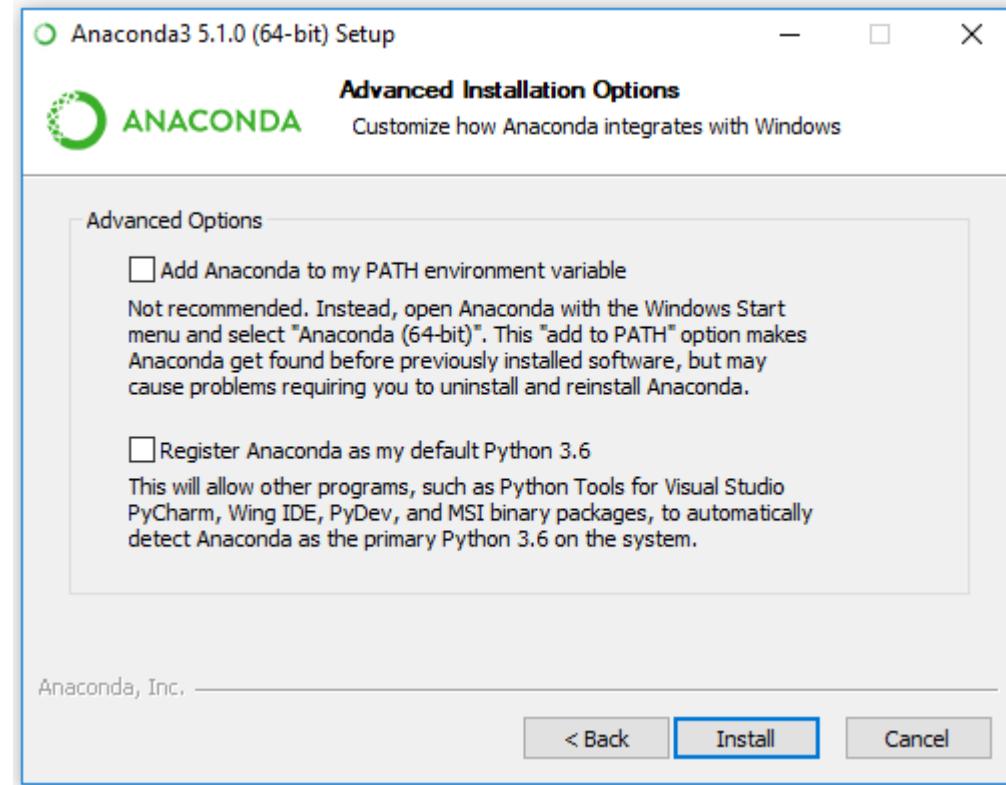
Installing Anaconda



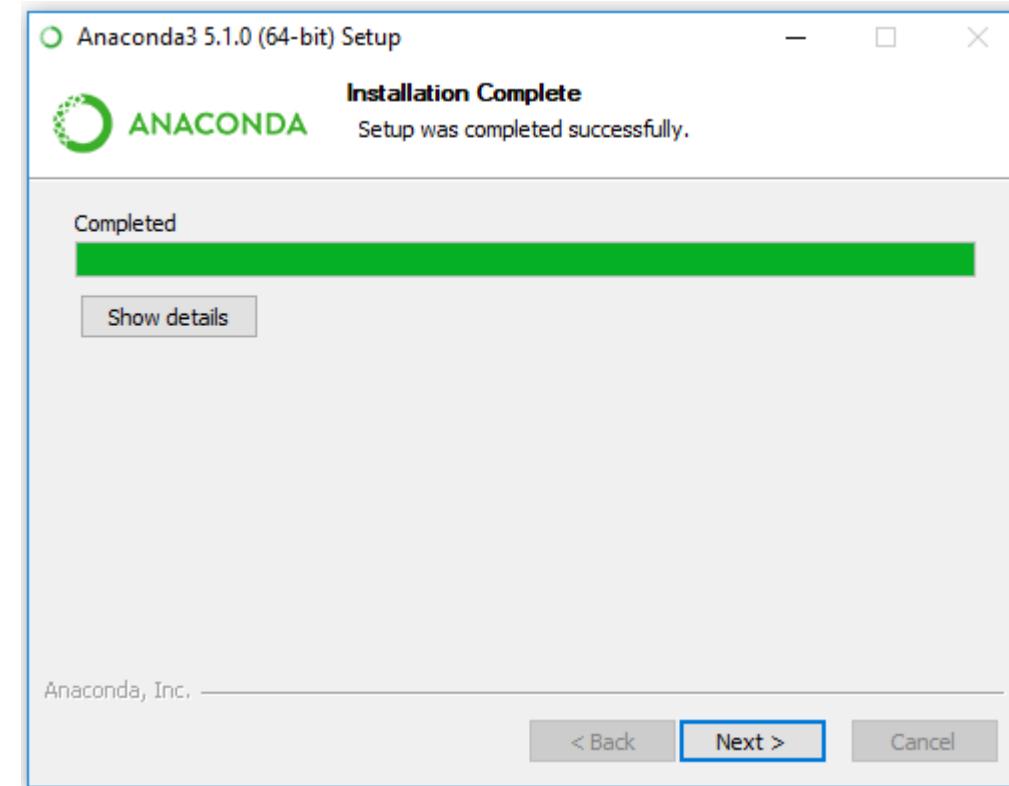
Installing Anaconda



Installing Anaconda

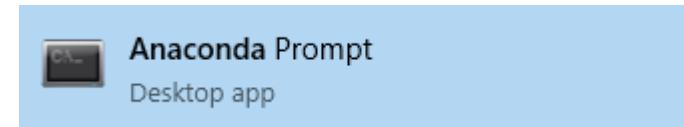


Installing Anaconda

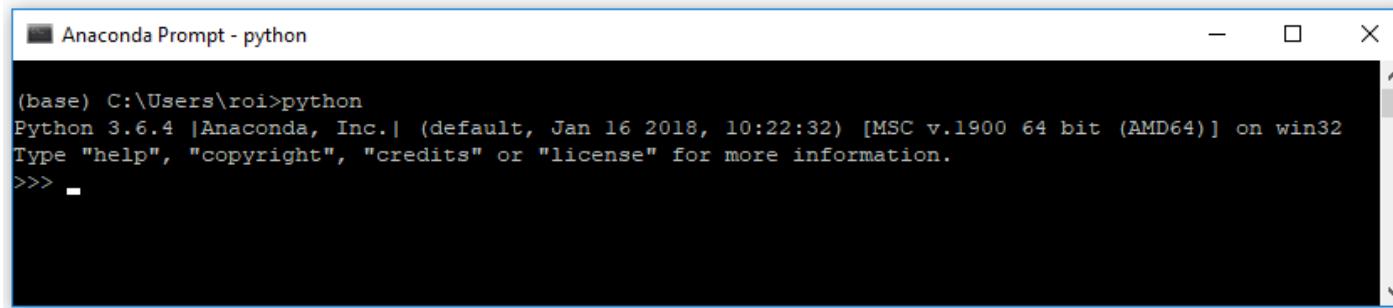


The Python Shell

- ▶ The Python shell is an interactive environment: the user enters Python statements that are executed immediately after the *Enter* key is pressed
- ▶ To start a Python shell from the command line, first open the Anaconda command prompt from the Window start menu:



- ▶ Then type python:

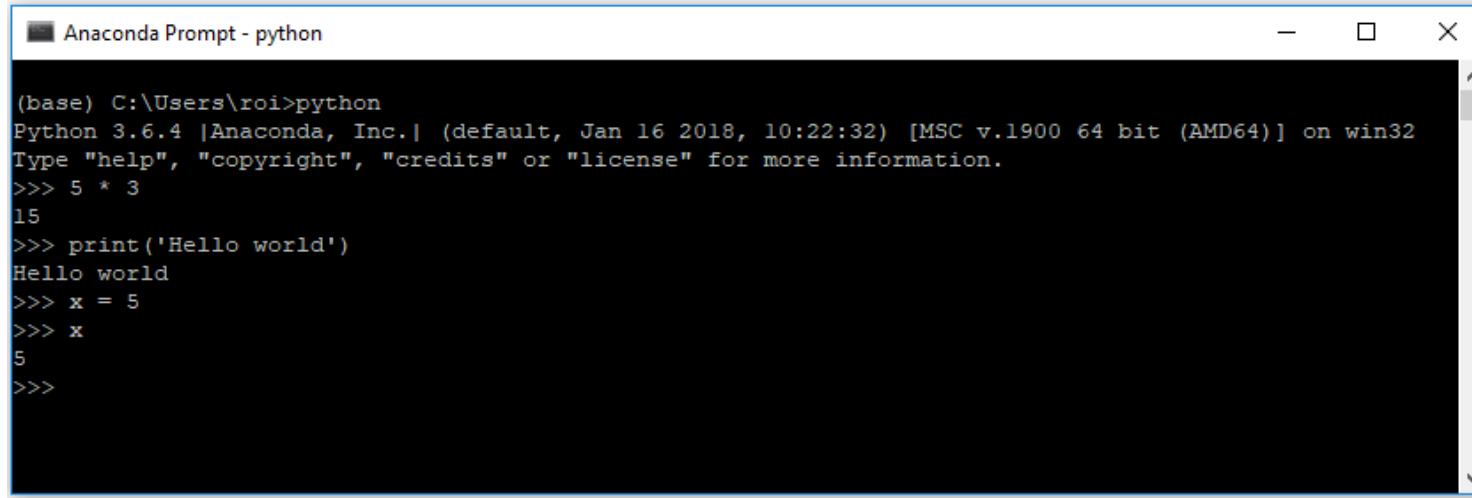


```
Anaconda Prompt - python
(base) C:\Users\roi>python
Python 3.6.4 |Anaconda, Inc.| (default, Jan 16 2018, 10:22:32) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> -
```

- ▶ The three chevrons (>>>) are the **prompt**, which is where you enter your Python commands

The Python Shell

- ▶ Enter the following calculations one by one and hit enter to get the result.



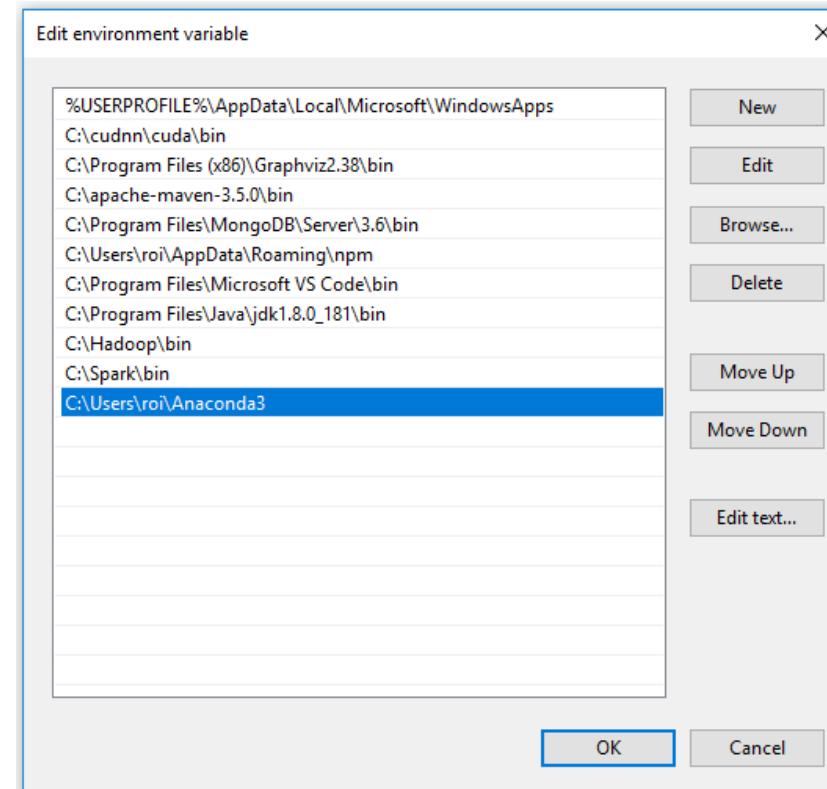
A screenshot of the Anaconda Prompt window titled "Anaconda Prompt - python". The window shows the Python 3.6.4 interactive shell. The session starts with the Python version and copyright information. Then, it performs a multiplication (5 * 3), prints "Hello world", assigns the value 5 to a variable x, and prints the value of x. Finally, it ends with a blank line. The window has a standard Windows-style title bar and scroll bars.

```
(base) C:\Users\roi>python
Python 3.6.4 |Anaconda, Inc.| (default, Jan 16 2018, 10:22:32) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 5 * 3
15
>>> print('Hello world')
Hello world
>>> x = 5
>>> x
5
>>>
```

- ▶ In Python, we use `print()` function to print something to the screen
- ▶ To exit the Python shell, type `exit()` or Ctrl-Z and hit Enter

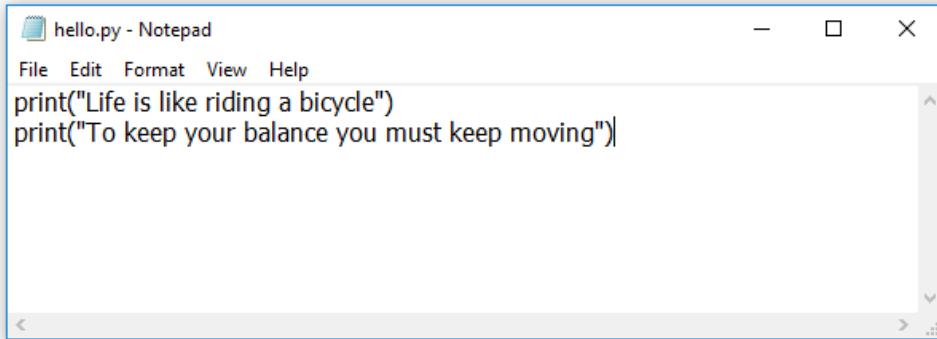
The Python Shell

- ▶ To enable running the Python shell from any location in your computer, you can add `python.exe` to your PATH environment variable:



Script Mode

- ▶ Python Shell is great for testing small chunks of code, but the statements you enter in the shell are not saved anywhere
- ▶ If you want to execute the same set of statements multiple times you would rather save the entire code in a script file (with extension .py), and then use the Python interpreter in script mode to execute the code in the file
- ▶ To create the script you can use any text editor
- ▶ For example, create a folder C:\Python, and create a new file hello.py in this folder
- ▶ Edit the file in Notepad and add the following code to it:

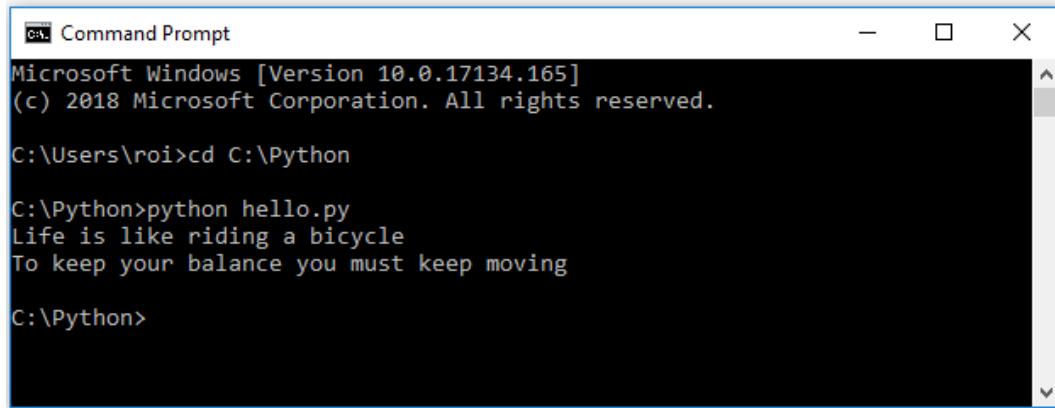


```
hello.py - Notepad
File Edit Format View Help
print("Life is like riding a bicycle")
print("To keep your balance you must keep moving")
```

Script Mode

- ▶ The file hello.py is called source file / script file / module
- ▶ To execute the program, open command prompt, change your current working directory to C:\Python using the cd command, then type the following command:

```
python hello.py
```
- ▶ This instructs the shell to invoke the Python interpreter, sending it the file hello.py as the script to execute
- ▶ Output from the program is then returned to the shell and displayed in your console

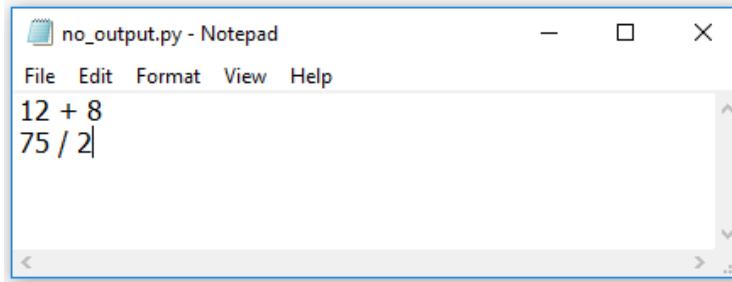


The screenshot shows a Microsoft Windows Command Prompt window titled "Command Prompt". The window title bar includes the text "Microsoft Windows [Version 10.0.17134.165]" and "(c) 2018 Microsoft Corporation. All rights reserved.". The main area of the window displays the following text:

```
C:\Users\roi>cd C:\Python
C:\Python>python hello.py
Life is like riding a bicycle
To keep your balance you must keep moving
C:\Python>
```

Script Mode

- ▶ To print values from a Python script you must explicitly use the `print()` function
- ▶ For example, create a new file named `no_output.py` with the following code:



A screenshot of a Windows Notepad window titled "no_output.py - Notepad". The window shows the following code:

```
12 + 8
75 / 2
```

- ▶ To run the file enter the following command:

```
C:\Python>python no_output.py
C:\Python>
```

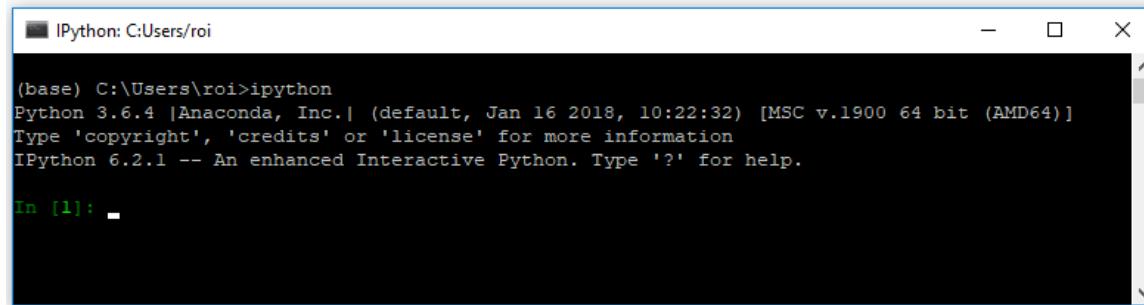
- ▶ As you can see, the program didn't output anything.

Exercise (1)

- ▶ Create a Python script in C:\Python named welcome_to_ds.py
- ▶ In the script print your name and age on two separate lines
- ▶ Run the script from the Python Shell

IPython Shell

- ▶ The IPython shell and the related interactive, browser-based IPython Notebook provide a powerful interface to the Python language
- ▶ IPython has several advantages over the native Python shell, including easy interaction with the operating system, introspection and tab completion
- ▶ IPython is included in the Anaconda distribution
- ▶ To start an interactive IPython session from the command line, simply type ipython:



A screenshot of a Windows command-line interface window titled "IPython: C:\Users\roi". The window displays the following text:
(base) C:\Users\roi>ipython
Python 3.6.4 |Anaconda, Inc.| (default, Jan 16 2018, 10:22:32) [MSC v.1900 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 6.2.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: -

- ▶ The prompt In [1]: is where you type your Python statements and replaces the native Python >>> shell prompt

IPython Shell

- ▶ The counter in square brackets increments with each Python statement or code block:

```
In [1]: 4 + 5
Out[1]: 9

In [2]: x = 10

In [3]: print(x)
10

In [4]: -
```

- ▶ To exit the IPython shell, type quit or exit (no parentheses are required)

Jupyter Notebook

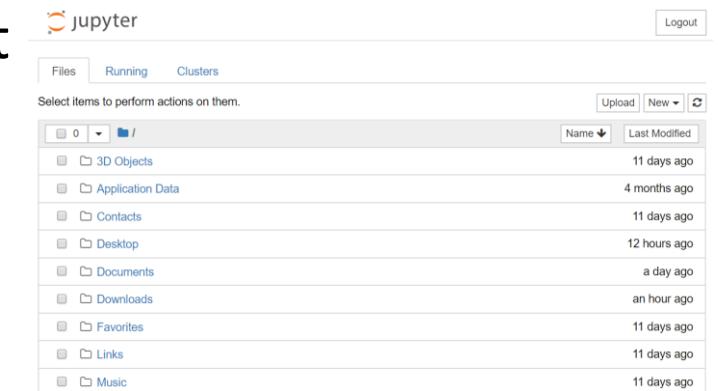
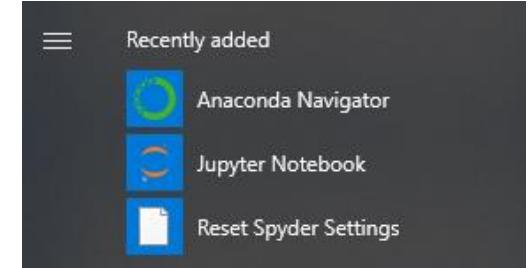
- ▶ **Jupyter Notebook** (formerly IPython Notebooks) provides an interactive environment for Python programming within a web browser
- ▶ Its main advantage over IPython shell is that Python code can be combined with documentation, rich text, images and even rich media such as embedded videos
- ▶ A **notebook document** is a JSON document, containing an ordered list of input/output cells, that ends with the “.ipynb” extension
 - ▶ Cells can contain code, text, mathematics, plots and rich media
- ▶ A **Jupyter kernel** is a program responsible for executing the code inside a notebook the and communicating the results back to the browser.
 - ▶ By default Jupyter Notebook ships with the IPython kernel, but it has bindings for other languages as well (such as Julia and R)
- ▶ If you've installed the Anaconda distribution, you already have Jupyter Notebook!

Starting the Jupyter Notebook Server

- ▶ Start the Jupyter Notebook server from the Windows-Start menu
- ▶ You can also start it from the command line by typing:

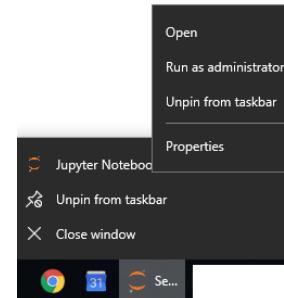
```
ipython notebook
```

- ▶ This will print some information about the notebook server in your terminal, and also open a browser window at the URL of the local IPython notebook application
- ▶ By default this is `http://127.0.0.1:8888`, though it will default to a different port if 8888 is in use



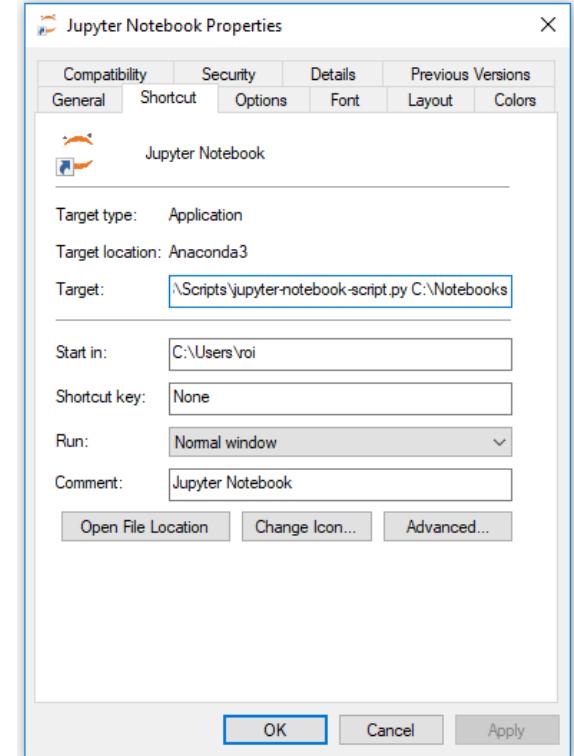
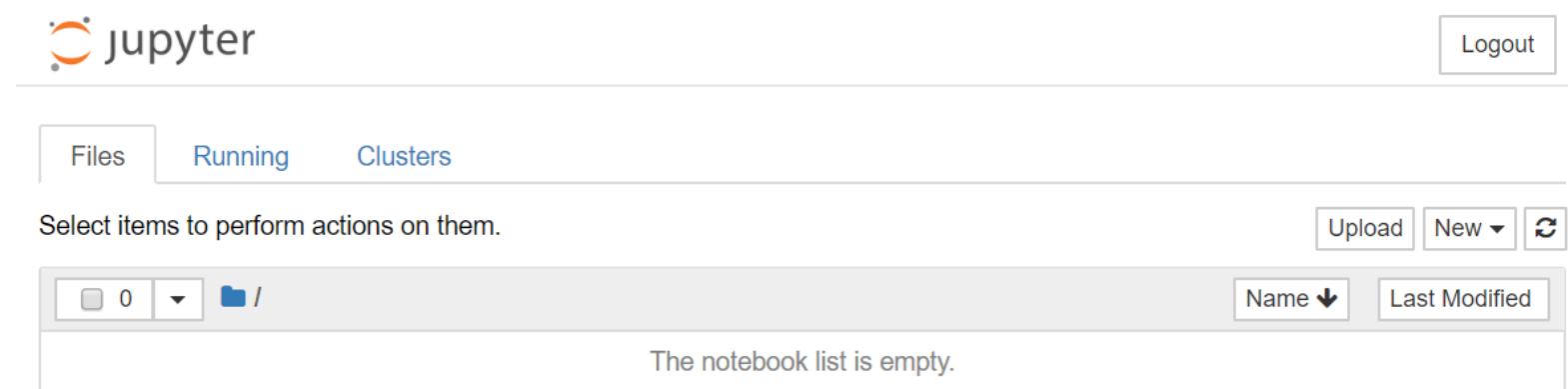
Starting the Jupyter Notebook Server

- ▶ The notebook index page contains a list of the notebooks currently available in the directory from which the notebook server was started
- ▶ This is also the default directory to which notebooks will be saved, so it is a good idea to run the notebook server somewhere convenient in your directory hierarchy for the project you are working on
- ▶ We'll change the startup directory for the notebook server to C:\Notebooks
- ▶ First create the directory C:\Notebooks
- ▶ Now change the shortcut to Jupyter Notebook to start the server from that folder
 - ▶ Right-click the Jupyter Notebook shortcut, and open its properties



Starting the Jupyter Notebook Server

- ▶ In the Target box remove %USERPROFILE% and type instead C:\Notebooks
- ▶ Click OK
- ▶ Close Jupyter Notebook and run it again
- ▶ Now you should see an empty workspace directory

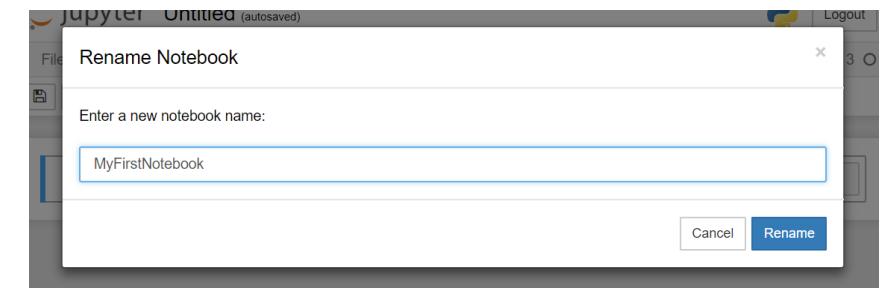
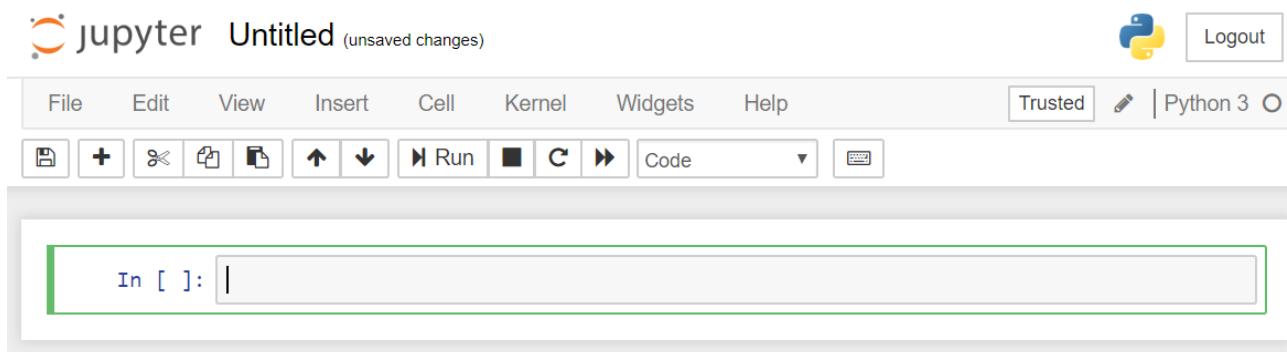
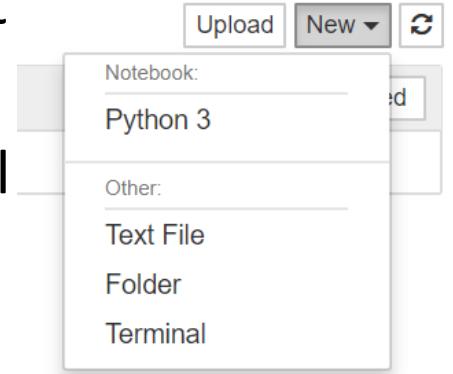


Notebook Index Page

- ▶ The index page contains three tabs:
 - ▶ **Files** lists the IPython notebooks and subdirectories within the current working directory
 - ▶ **Running** lists those notebooks that are currently active within your session (even if they are not open in a browser window)
 - ▶ **Clusters** provides an interface to IPython's parallel computing engine
- ▶ From the index page, you can start a new notebook (by clicking on “New Notebook”) or open an existing notebook (by clicking on its name)
- ▶ To import an existing notebook into the index page, drag the notebook file into the index listing from elsewhere on your operating system
- ▶ To stop the notebook server, press CTRL-C in the terminal window it was started from

Create a New Notebook

- ▶ To start a new notebook, click the “New Notebook” button and select Python 3
- ▶ This opens a new browser tab containing the interface where you will write your code and connects it to an IPython kernel
- ▶ In the title bar the name of the first notebook you open will probably be “Untitled”
- ▶ Click on it to rename it to something more informative
 - ▶ This will also rename the .ipynb file where the notebook is saved



Cell Types

- ▶ There are four types of input cells where you can write the content for your notebook:
 - ▶ **Code cells:** the default type of cell, this type of cell consists of executable code
 - ▶ **Markdown cells:** this type of cell allows for a rich form of documentation for your code
 - ▶ When executed, the input to a markdown cell is converted into HTML, which can include mathematical equations, font effects, lists, tables, embedded images and videos
 - ▶ **Raw cells:** input into this type of cell is not changed by the notebook – its content and formatting is preserved exactly

Running Cells

- ▶ Each cell can consist of more than one line of input, and the cell is not interpreted until you “run” it
- ▶ This is achieved either by selecting the appropriate option from the menu bar (under the “Cell” drop-down submenu), by clicking the “play” button on the tool bar, or through the following keyboard shortcuts:
 - ▶ **Shift-Enter:** Execute the cell, showing any output, and then *move the cursor* onto the cell below. If there is no cell below, a new, empty one will be created.
 - ▶ **CTRL-Enter:** Execute the cell in place, but *keep the cursor* in the current cell. Useful for quick “disposable” commands to check if a command works.
 - ▶ **Alt-Enter:** Execute the cell, showing any output, and then *insert and move the cursor to a new cell* immediately beneath it.

Code Cells

- ▶ All cells start as “Code” cells by default
- ▶ You can enter anything into a code cell that you can when writing a Python program in an editor or at the regular IPython shell
- ▶ Code in a given cell has access to objects defined in other cells (providing they have been run)
- ▶ For example,

```
In [ ]: x = 10
```

- ▶ Pressing Shift-Enter or clicking Run Cell executes this statement (defining x but producing no output) and opens a new cell underneath the old one:

```
In [1]: x = 10
```

```
In [ ]: |
```

Code Cells

- ▶ Enter the following statement at this new prompt:

```
In [1]: x = 10
```

```
In [ ]: print("x = ", x)
```

- ▶ and executing as before produces output and opens a third empty input cell:

```
In [1]: x = 10
```

```
In [2]: print("x =", x)
```

```
x = 10
```

```
In [ ]: |
```

- ▶ You can edit the value of x in input cell 1 and rerun the entire document to update the output by choosing Kernel -> Restart & Run all

Code Cells

- ▶ It is also possible to set a new value for `x` *after the calculation* in cell 2:

```
In [3]: x = 5
```

- ▶ Running cell 3 and then cell 2 then changes the output of cell 2 to:

```
In [4]: print("x =", x)
```

```
x = 5
```

- ▶ even though the cell above still defines `x` to be 10
- ▶ Unless you run the entire document from the beginning, the output doesn't necessarily reflect the output of a script corresponding to the code cells taken in order

Edit Mode

- ▶ There are two modes of operation in a notebook: edit mode and command mode
- ▶ The keyboard does different things depending on which mode the notebook is in
- ▶ **Edit mode** is indicated by a green cell border and a prompt showing in the editor area:



```
In [1]: x = 10
```

- ▶ When a cell is in edit mode, you can type into the cell, like a normal text editor
- ▶ In edit mode the arrow keys navigate the *contents* of the cell
- ▶ Enter edit mode by pressing Enter or using the mouse to click on a cell's editor area

Command Mode

- ▶ Command mode is indicated by a grey cell border with a blue left margin:



```
In [1]: x = 10
```

- ▶ When you are in command mode, you are able to edit the notebook as a whole, but not type into individual cells
- ▶ In command mode, the keyboard is mapped to a set of shortcuts that let you perform notebook and cell actions efficiently
- ▶ For example, pressing `a` in command mode will create a new cell above the current cell, and arrow keys navigate *through the cells*
- ▶ Enter command mode by pressing `Esc` or using the mouse to click *outside* a cell's editor area

Keyboard Shortcuts

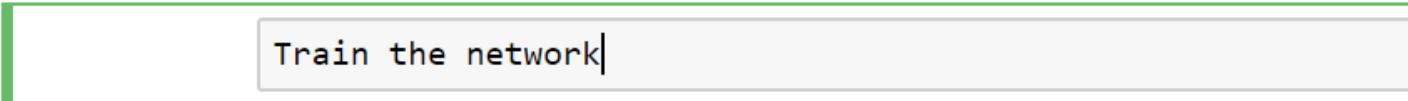
- ▶ The Help->Keyboard Shortcuts dialog lists the available shortcuts
- ▶ Useful commands in command mode:
 - ▶ Basic navigation: shift-enter, ctrl-enter, up/k, down/j
 - ▶ Saving the notebook: s
 - ▶ Change cell type: y, m, 1-6, t
 - ▶ Insert cell above: a
 - ▶ Insert cell below: b
 - ▶ Copy cell: c
 - ▶ Paste cell: v
 - ▶ Delete selected cell: dd

Exercise (2)

- ▶ Create a new notebook in C:\Notebooks named WelcomeToDS.ipynb
- ▶ Print your name and age in two separate cells
- ▶ Insert a new cell between the two existing cells
- ▶ In the new cell print the current date
- ▶ Copy the first two cells to another notebook named SecondNotebook.ipynb
- ▶ Add a third cell that prints your favorite color
- ▶ Delete the second cell
- ▶ Run the second notebook from start to end
- ▶ Delete the second notebook

Markdown Cells

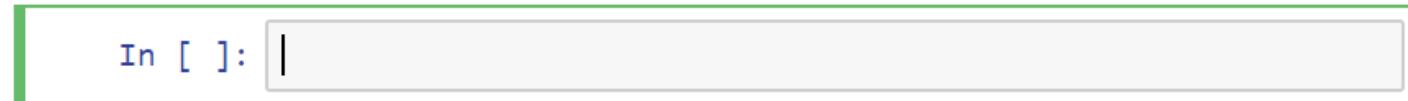
- ▶ Markdown cells are useful to help document your Notebook or to write detailed “readme” descriptions within your notebook
- ▶ You can change the cell type from the menu Cell > Cell Type or by using the key shortcut m (in command mode)



```
Train the network
```

- ▶ When you “run” the markdown cell, the text in the cell is converted into HTML:

```
Train the network
```



```
In [ ]: |
```

- ▶ Markdown is a popular markup language that is a superset of HTML
 - ▶ Its specification can be found here: <https://daringfireball.net/projects/markdown/>

Markdown Basics

- ▶ You can make text italic or bold by using asterisks or underscores:

```
Surrounding text by two asterisks denotes **bold style**;  
using one asterisk denotes *italic text*, as does _a single  
underscore_.
```

Surrounding text by two asterisks denotes **bold style**; using one asterisk denotes *italic text*, as does a single underscore.

- ▶ Inline code examples are created by surrounding the text with backticks (`):

```
Here are some Python keywords: `for` , `while` , and `lambda` .
```

Here are some Python keywords: `for` , `while` , and `lambda` .

- ▶ New paragraphs are started after a blank line

Headings

- ▶ You can add headings by starting a line with one (or multiple) # followed by a space, as in the following example:

```
# Heading 1
## Heading 1.1
### Heading 1.1.1
```

Heading 1

Heading 1.1

Heading 1.1.1

- ▶ Six levels of heading are supported

Links

- ▶ Links are created by providing a URL in round brackets after the text to be turned into a link in square brackets

Here is a link to the [\[Python website\]](http://python.org)(<http://python.org>)

Here is a link to the [Python website](#)

- ▶ If the link is to a file on your local system, give as the URL the path, relative to the notebook directory, prefixed with files/:

Here is [\[the housing data file\]](#)([files/datasets/housing.csv](#)).

Here is [the housing data file](#).

Lists

- ▶ Itemized lists are created using any of the markers *, + or -, and nested sublists are simply indented:

The inner planets and their satellites:

```
* Mercury
* Venus
* Earth
  * The Moon
* Mars
  * Phobus
  * Deimos
```

The inner planets and their satellites:

- Mercury
- Venus
- Earth
 - The Moon
- Mars
 - Phobus
 - Deimos

- ▶ Ordered (numbered) lists are created by preceding items by a number followed by a full stop (period) and a space:

1. Symphony No. 1 in C major, Op. 21
2. Symphony No. 2 in D major, Op. 36
3. Symphony No. 3 in E-flat major ("Eroica"), Op. 55

1. Symphony No. 1 in C major, Op. 21
2. Symphony No. 2 in D major, Op. 36
3. Symphony No. 3 in E-flat major ("Eroica"), Op. 55

Mathematical Equations

- ▶ Mathematical equations can be written in LaTeX
- ▶ Inline expressions can be added by surrounding the latex code with \$:

```
Defining the function $f(x) = \sin(x^2)$
```

Defining the function $f(x) = \sin(x^2)$

- ▶ Expressions on their own line are surrounded by \$\$:

```
$$\int_0^\infty e^{-x^2} dx = \frac{\sqrt{\pi}}{2}$$
```

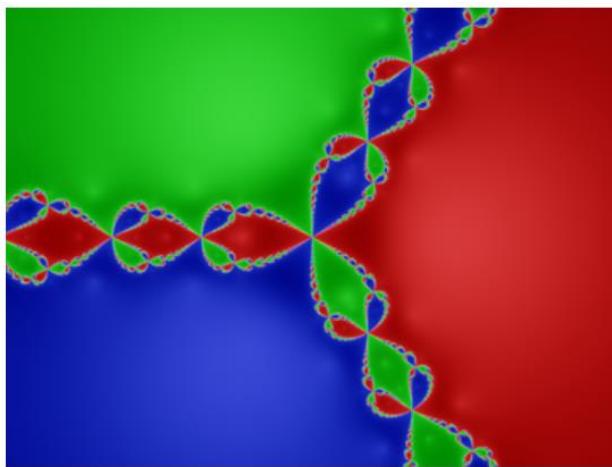
$$\int_0^\infty e^{-x^2} dx = \frac{\sqrt{\pi}}{2}$$

- ▶ Learn LaTeX in 30 minutes:
[https://www.sharelatex.com/learn/Learn LaTeX in 30 minutes](https://www.sharelatex.com/learn/Learn_LaTeX_in_30_minutes)

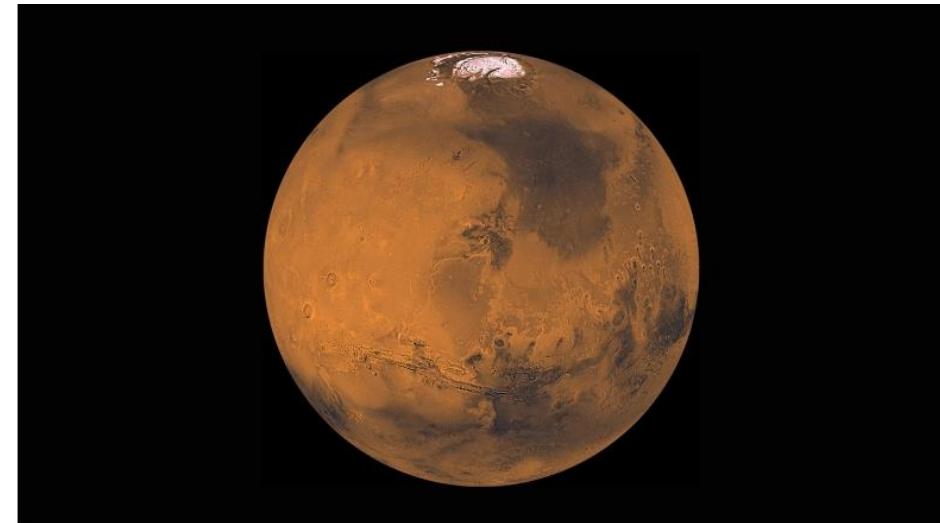
Images

- ▶ Links to image files work in exactly the same way as ordinary links, but are preceded by an exclamation mark, !
- ▶ The text in square brackets between the exclamation mark and the link acts as *alt text* to the image

![An interesting plot of the Newton fractal].
(/files/images/newton_fractal.png)



![Planet Mars].
(<https://boygeniusreport.files.wordpress.com/2016/05/mars.jpg?quality=98&strip=all&w=782>)



General HTML

- The markdown used by IPython notebooks encompasses HTML, so valid HTML entities and tags can be used directly, including CSS styles

```
The following <em>Punnett table</em> is <span style="text-decoration: underline">marked up</span> in HTML.


|        |    | Male   |    |
|--------|----|--------|----|
|        |    | Female |    |
| Male   | A  | a      | aa |
|        | a  | Aa     | aa |
| Female | Aa | Aa     | aa |

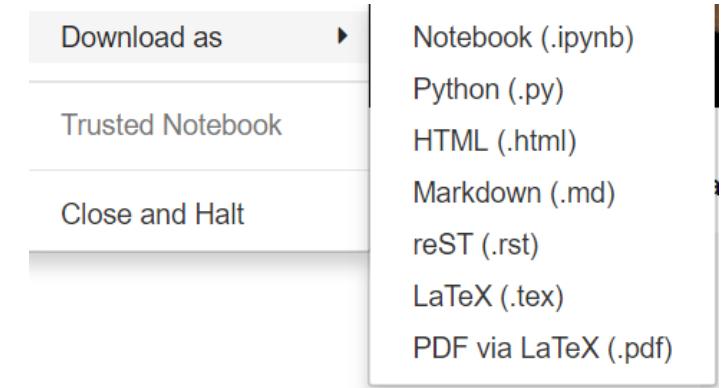

```

The following *Punnett table* is marked up in HTML.

		Male	
Male	Female		
	A	a	aa
Male	A	a	aa
Female	a	Aa	aa
	a	Aa	aa

Converting Notebooks to Other Formats

- ▶ nbconvert is a tool, installed with Jupyter notebook, to convert notebooks from their native .ipynb format to any of several alternative format
- ▶ You can invoke this tool from the File > Download as menu



- ▶ Or run it from the command line as:

```
ipython nbconvert --to <format> <notebook.ipynb>
```

- ▶ where notebook.ipynb is the name of the notebook file to be converted and format is the desired output format
- ▶ The default (if no format is given), is to produce a static HTML file

Converting to Python

- ▶ Choose File > Download as > Python (.py)
- ▶ This will save the notebook as *MyFirstNotebook.py* in your Download folder
- ▶ If any of the notebook's code cells contain IPython magic functions, this script may only be executable from within an IPython session
- ▶ Markdown and other text cells are converted to comments in the generated Python script code
- ▶ You can now run this script from the Python shell

Converting to Python

```
# coding: utf-8

# In[1]:


x = 10


# In[2]:


print("x =", x)

# In[3]:


x = 5


# Train the network

# Surrounding text by two asterisks denotes **bold style**;
# using one asterisk denotes *italic text*, as does _a single
# underscore_.
```

```
C:\Users\roi>cd c:\Python

c:\Python>python MyFirstNotebook.py
x = 10

c:\Python>
```

Exercise (3)

- ▶ Create the following notebook using markdown cells:

Fruit

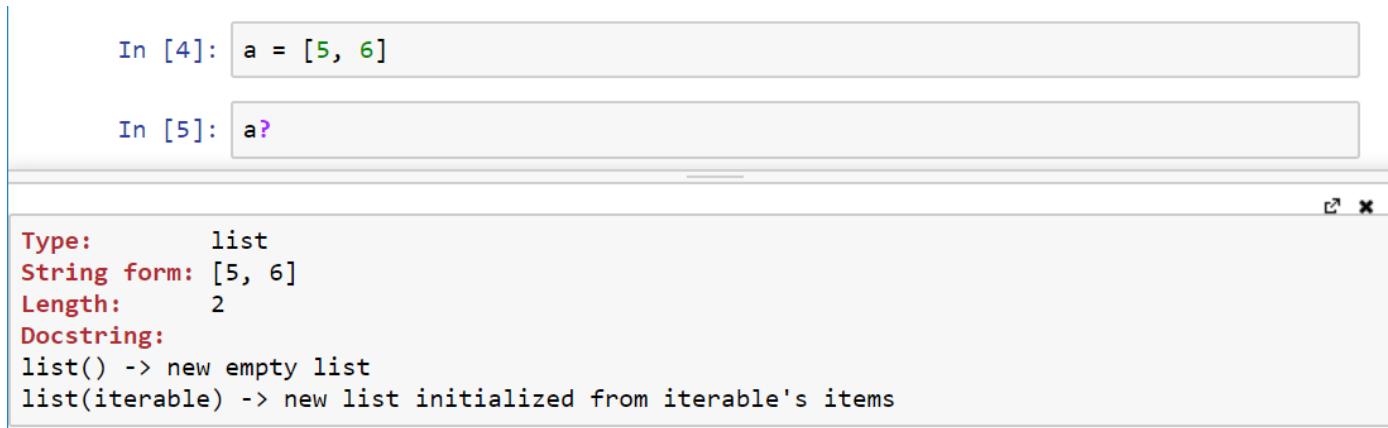
- Pears
- Apples
 - Granny Smith
 - Gala
- Oranges
- Berries
 - Strawberries
 - Blueberries
 - Raspberries
- Bananas



$$\sum_{i=1}^n = \frac{n(n+1)}{2}$$

Help Commands

- ▶ There are various helpful commands to obtain information:
 - ▶ Typing a single ‘?’ outputs an overview of the usage of IPython’s main features
 - ▶ %quickref provides a brief reference summary of each of the main IPython commands and “magics”
 - ▶ help() or help(object) invokes Python’s own help system (interactively or for object if specified)
 - ▶ Typing one question mark after an object name provides information about that object
- ▶ Example for introspecting an object:



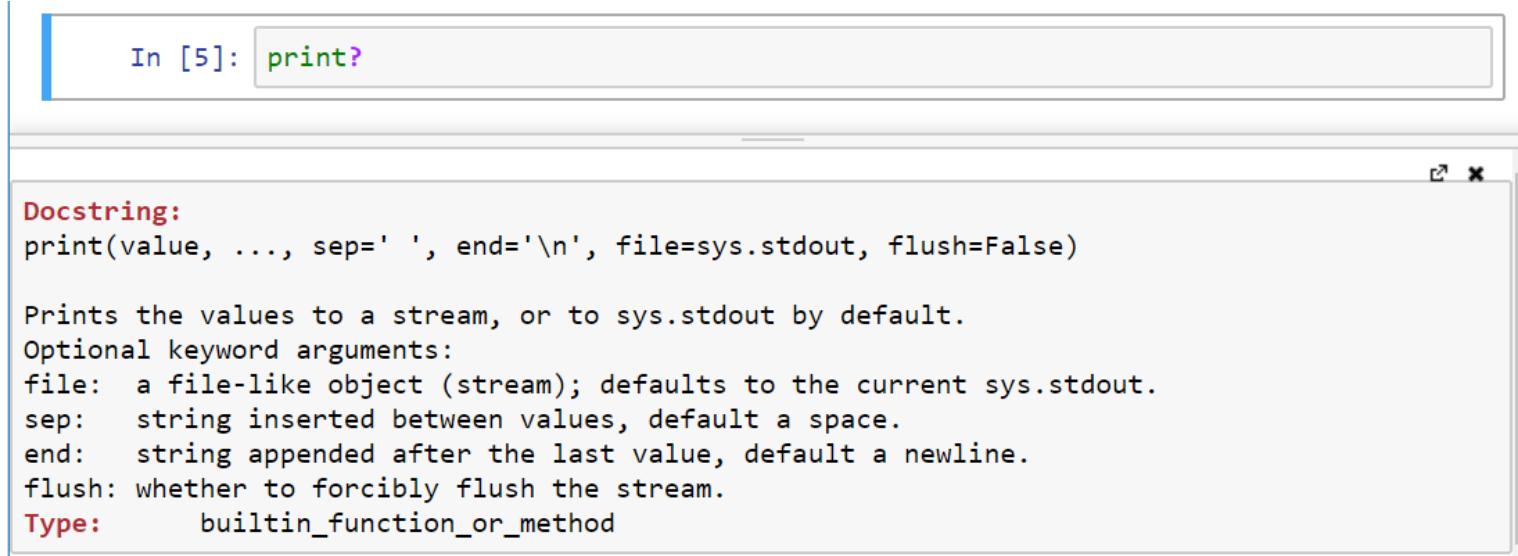
In [4]: a = [5, 6]

In [5]: a?

Type: list
String form: [5, 6]
Length: 2
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items

Help Commands

- ▶ The ? syntax is particularly useful as a reminder of the arguments that a function or method takes



In [5]: `print?`

Docstring:

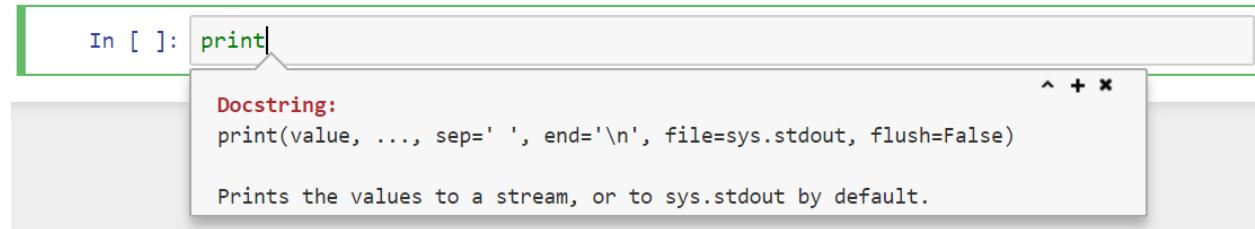
```
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file: a file-like object (stream); defaults to the current sys.stdout.
sep: string inserted between values, default a space.
end: string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
Type: builtin_function_or_method
```

- ▶ For some objects, the syntax `object??` returns more advanced information, such as the location and details of its source code.

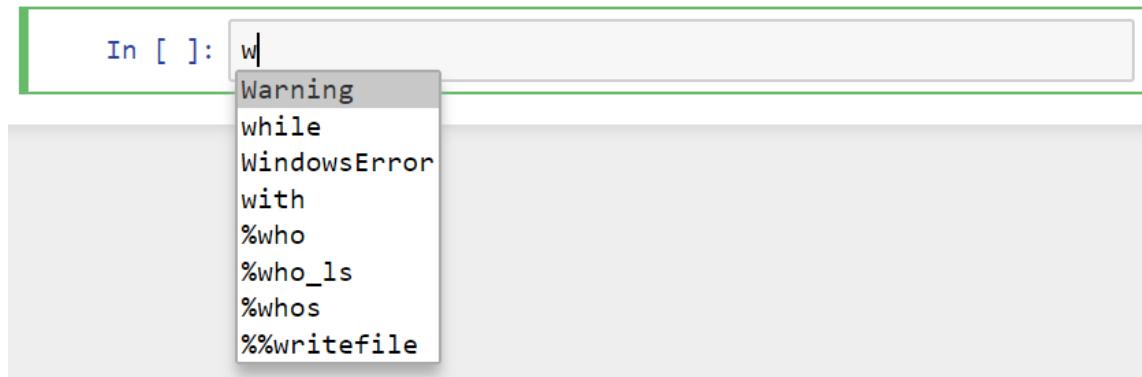
Help Commands

- ▶ Pressing Shift-Tab shows a tooltip containing additional information about the object or function



Tab Completion

- ▶ Just as with many command line shells, IPython supports tab completion: start typing the name of an object or keyword, press the <TAB> key, and it will autocomplete it for you or provide a list of options if more than one possibility exists
- ▶ For example, type the letter w and then press TAB:



- ▶ If you resume typing until the word becomes unambiguous (e.g., add the letters hi) and then press <TAB> again: it will be autocompleted to while.

History

- ▶ IPython stores both the commands you enter and the output they produce in the special variables **In** and **Out**
 - ▶ These are, in fact, a list and a dictionary respectively, and correspond to the prompts at the beginning of each input and output

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

```
John
```

```
In [2]: age = 25  
       print(age)
```

```
25
```

```
In [3]: In[1]
```

```
Out[3]: 'name = "John"\nprint(name)'
```

```
In [4]: exec(In[1])
```

```
John
```

In[1] simply holds the string version of the Python statements that were entered at index 1

To actually execute the statements, we must send the cell to Python's exec() built-in

History

- ▶ The Out[] dictionary is useful if you want to interact with past results
- ▶ For example:

```
In [5]: import math
```

```
In [6]: math.sin(2)
```

```
Out[6]: 0.9092974268256817
```

```
In [7]: math.cos(2)
```

```
Out[7]: -0.4161468365471424
```

```
In [8]: Out[6] ** 2 + Out[7] ** 2
```

```
Out[8]: 1.0
```

- ▶ Note that not all operations have outputs: for example, import statements and print statements don't affect the output

History

- ▶ There are a few more shortcuts:
 - ▶ `_iN` is the same as `In[N]`
 - ▶ `_N` is the same as `Out[N]`
 - ▶ The two most recent outputs are returned by the variables `_` and `__` respectively (skipping any commands with no output)

```
In [8]: Out[6] ** 2 + Out[7] ** 2
```

```
Out[8]: 1.0
```

```
In [9]: _
```

```
Out[9]: 1.0
```

Suppressing the Output

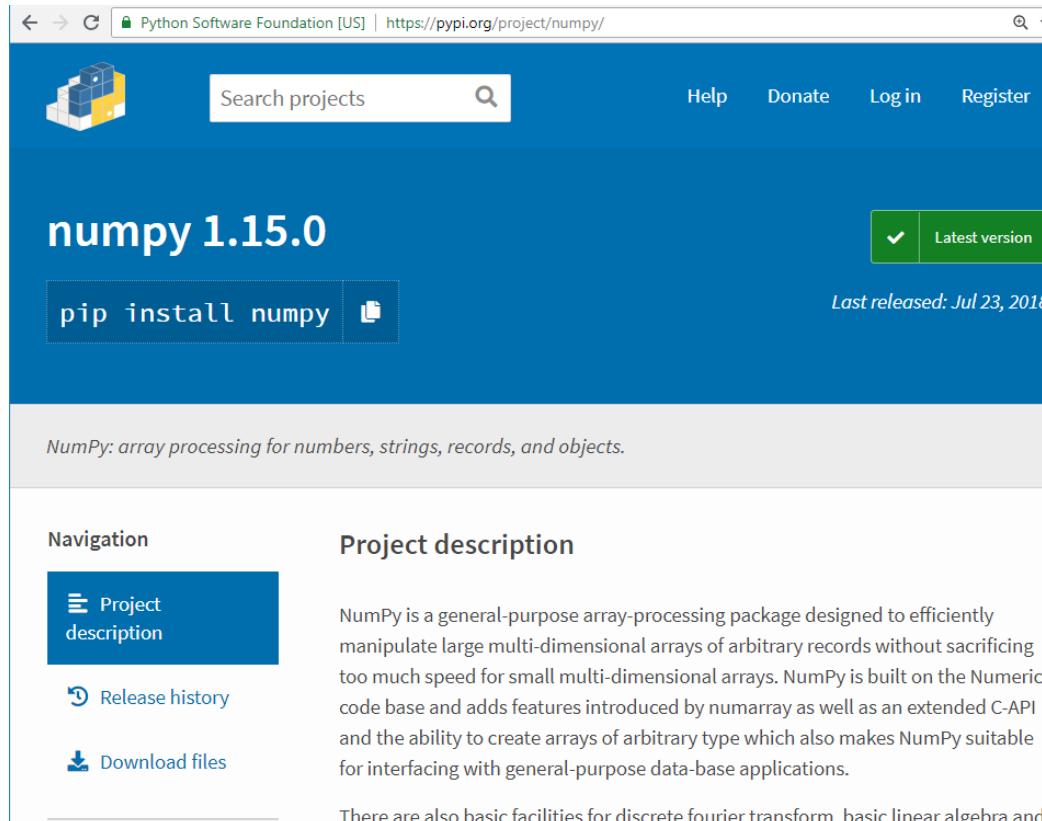
- ▶ Sometimes you might wish to suppress the output of a statement (this is perhaps most common with plotting commands)
- ▶ The easiest way to suppress the output of a command is to add a semicolon to the end of the line:

```
math.sin(2) + math.cos(2);
```

- ▶ Note that the result is computed silently, and the output is neither displayed on the screen or stored in the Out dictionary

PyPI

- ▶ The Python Package Index (PyPI) is a repository of software for the Python programming language
- ▶ <https://pypi.org/>



pip

- ▶ pip is a Python package management system used to install and manage software packages written in Python
- ▶ pip supports installing packages from PyPI, version control, local projects, and directly from distribution files
- ▶ To install the latest version of a package, type from the command-line:

```
pip install some-package
```

- ▶ To install a specific version:

```
pip install some-package==1.4
```

- ▶ To install a version that's “compatible” with a certain version:

```
pip install some-package~=1.4.2
```

- ▶ This means to install any version “==1.4.*” version that's also “>=1.4.2”.

pip

- ▶ To upgrade an already installed SomeProject to the latest from PyPI:

```
pip install --upgrade some-package
```

or: `pip install -u some-package`

- ▶ This command will additionally upgrade all the package requirements to the latest versions available

- ▶ To uninstall a package:

```
pip uninstall some-package
```

- ▶ To list all the packages installed:

```
pip list
```

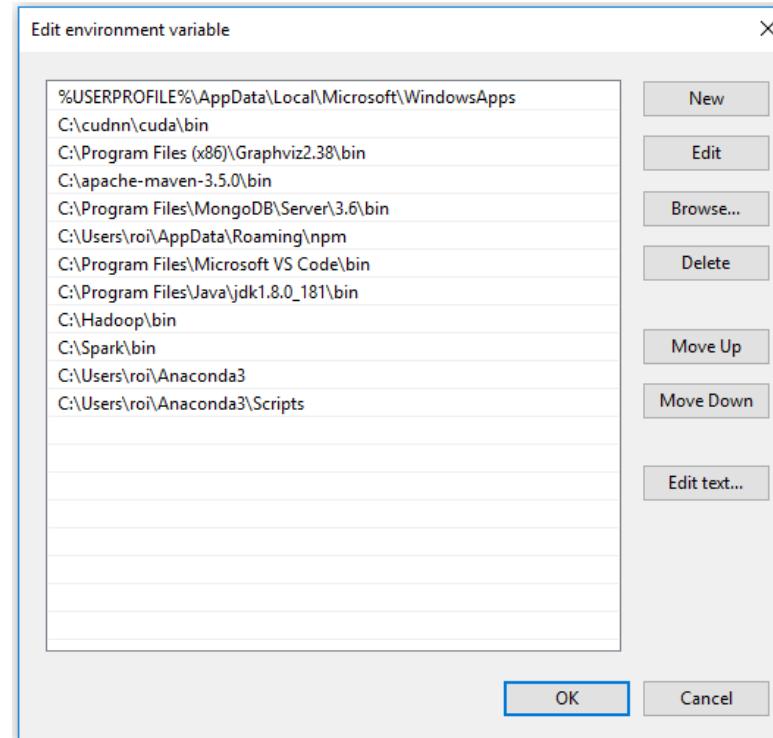
- ▶ To get information on an installed package (its version, dependencies, etc.):

```
pip show some-package
```

```
(base) C:\Users\roi>pip show numpy
Name: numpy
Version: 1.14.0
Summary: NumPy: array processing for numbers, strings, records, and objects.
Home-page: http://www.numpy.org
Author: NumPy Developers
Author-email: numpy-discussion@python.org
License: BSD
Location: c:\users\roi\anaconda3\lib\site-packages
Requires:
Required-by: tensorflow-gpu, tensorboard, tables, seaborn, PyWavelets, patsy,
pandas, odo, numexpr, numba, matplotlib, h5py, datashape, Bottleneck, bokeh, b
kcharts, astropy
```

pip

- ▶ pip.exe is typically located in the folder Scripts under your Anaconda installation folder
- ▶ To run pip from anywhere on your computer, you can add its path to the PATH environment variable:



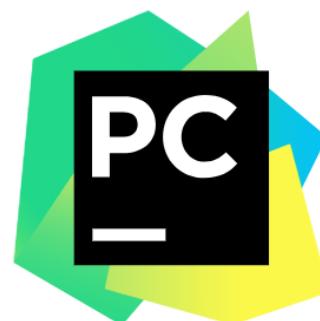
PyCharm

- ▶ PyCharm is an IDE for the Python language developed by JetBrains, a team responsible for one of the most famous Java IDE, the IntelliJ IDEA
- ▶ PyCharm is cross-platform, with Windows, macOS and Linux versions
- ▶ PyCharm integrates with IPython Notebook, has an interactive Python console, and supports Anaconda as well as multiple scientific packages including matplotlib and NumPy
- ▶ It provides code analysis, a graphical debugger, an integrated unit tester, integration with version control systems like git, and supports web development
- ▶ It provides a free Community Edition



Installing PyCharm

- ▶ Go to <https://www.jetbrains.com/pycharm/download>
- ▶ Choose your operating system
- ▶ Download the Community edition



Version: 2018.2
Build: 182.3684.100
Released: July 25, 2018

[System requirements](#)
[Installation Instructions](#)
[Previous versions](#)

Download PyCharm

[Windows](#) [macOS](#) [Linux](#)

Professional

Full-featured IDE
for Python & Web
development

[DOWNLOAD](#)

Free trial

Community

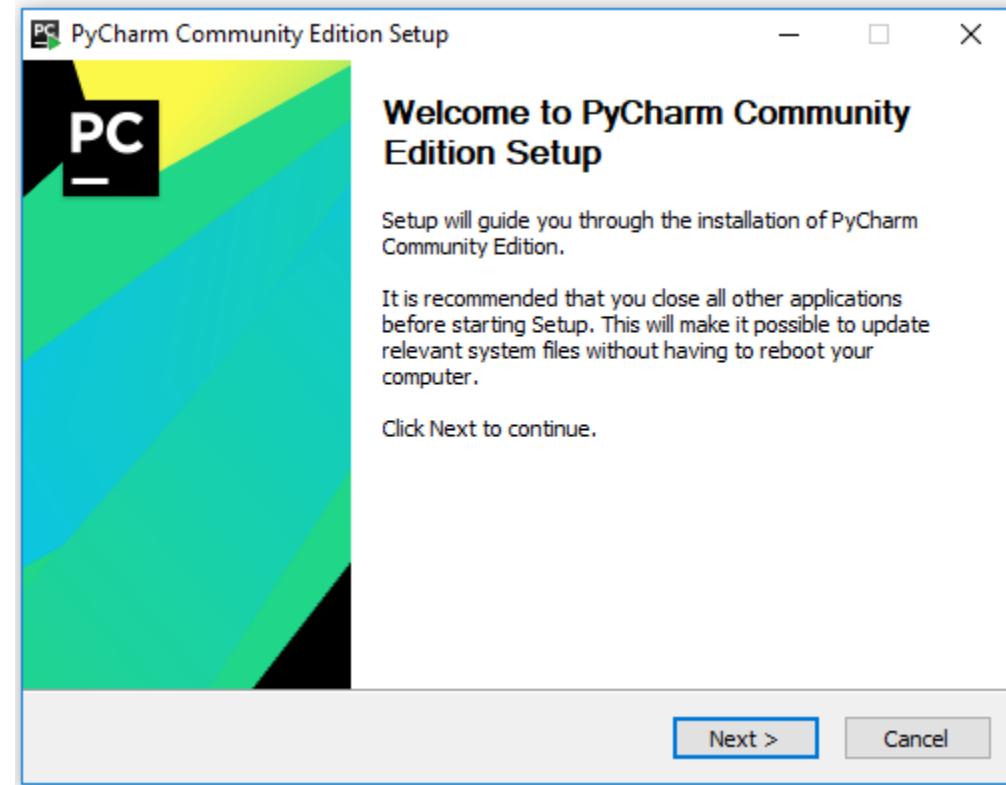
Lightweight IDE
for Python & Scientific
development

[DOWNLOAD](#)

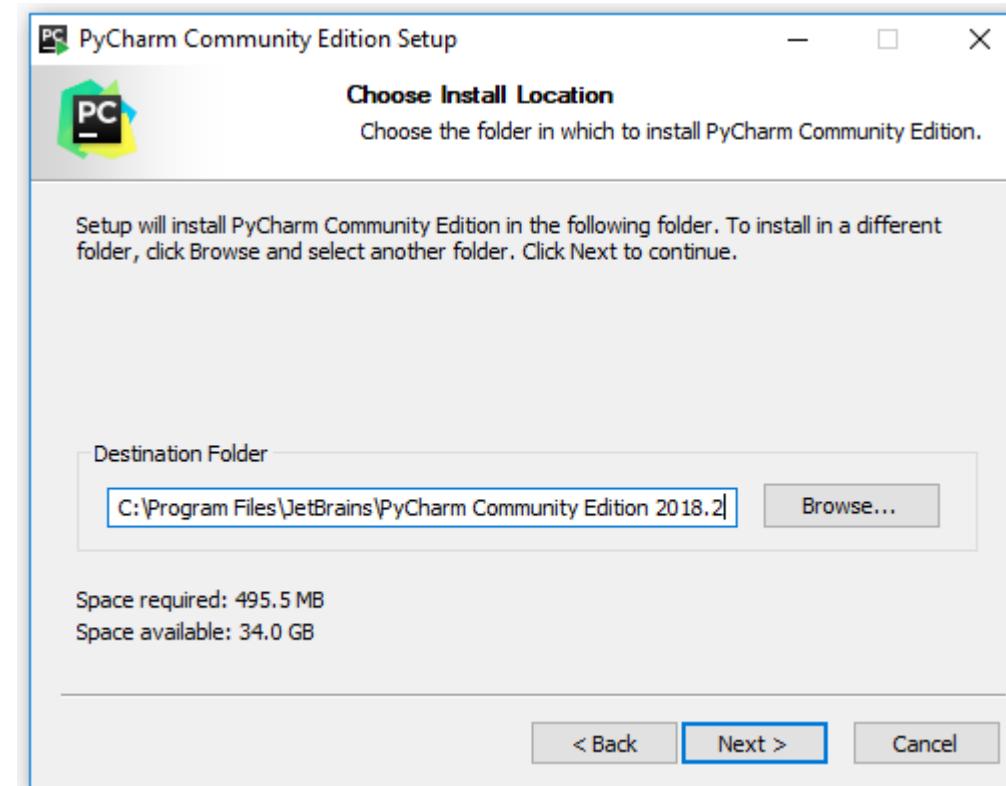
Free, open-source

Installing PyCharm

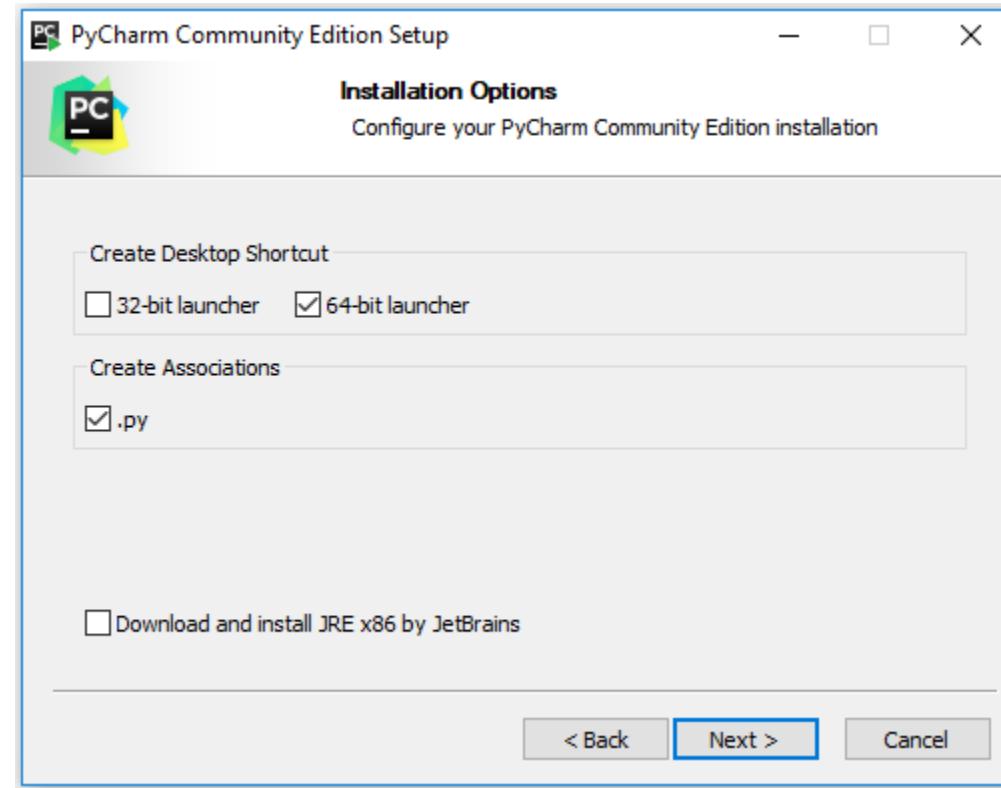
- ▶ Click Next to install



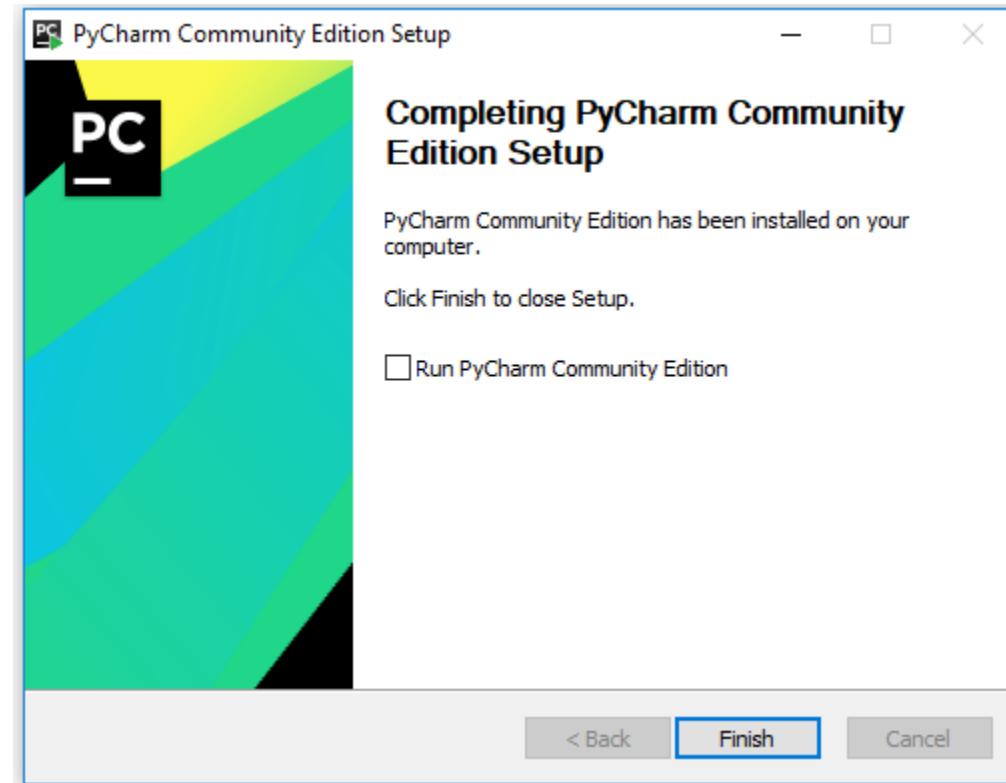
Installing PyCharm



Installing PyCharm

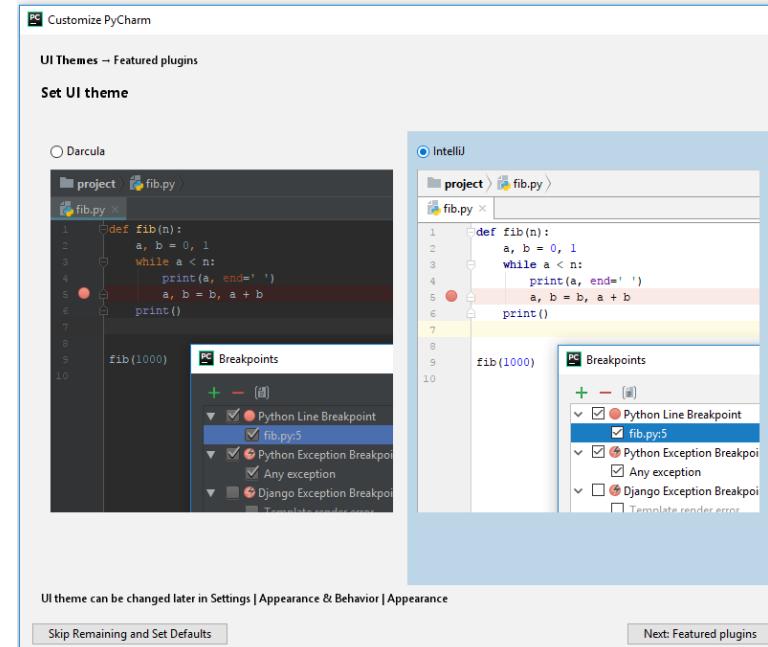


Installing PyCharm



Installing PyCharm

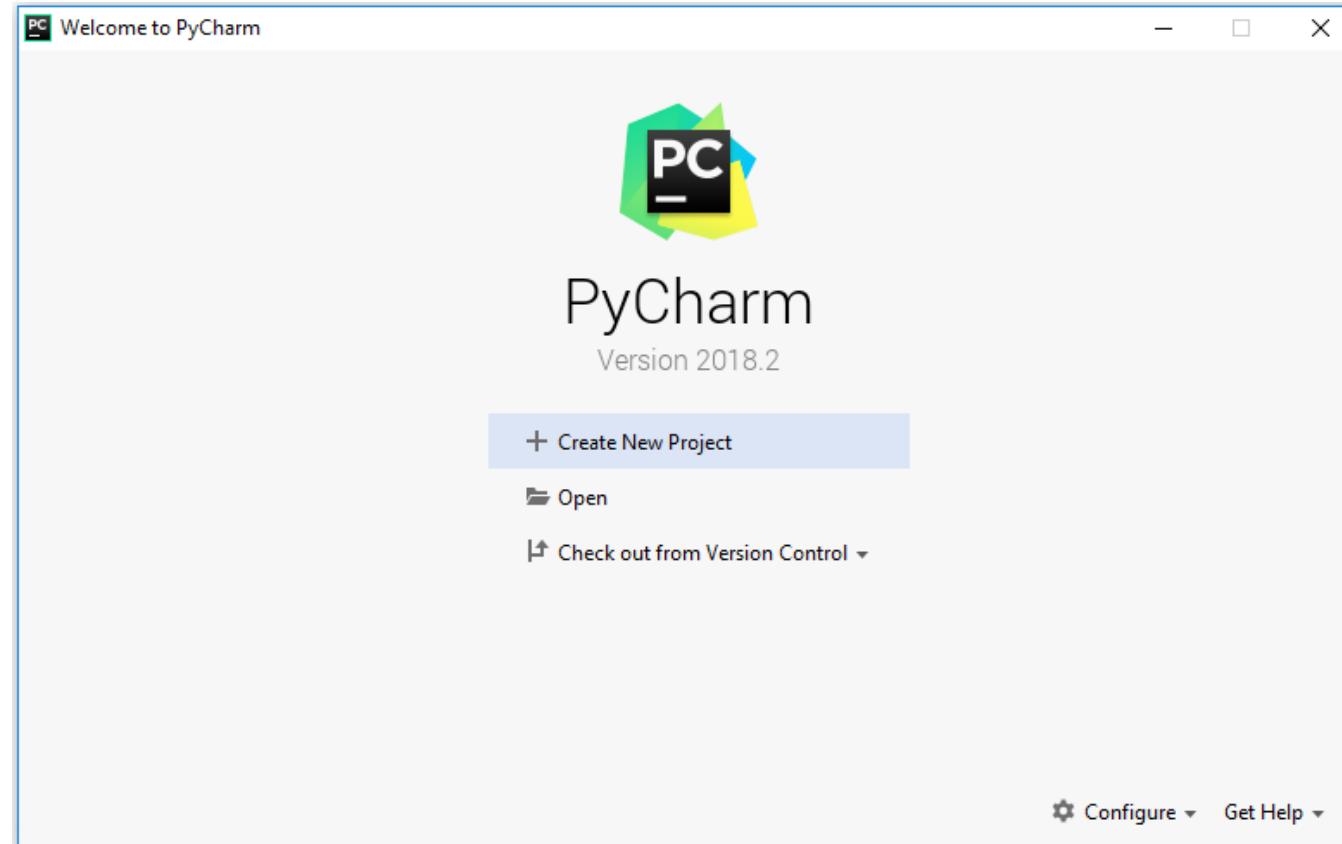
- ▶ Start PyCharm
- ▶ Choose your favorite UI theme



- ▶ Click Skip Remaining and Set Defaults

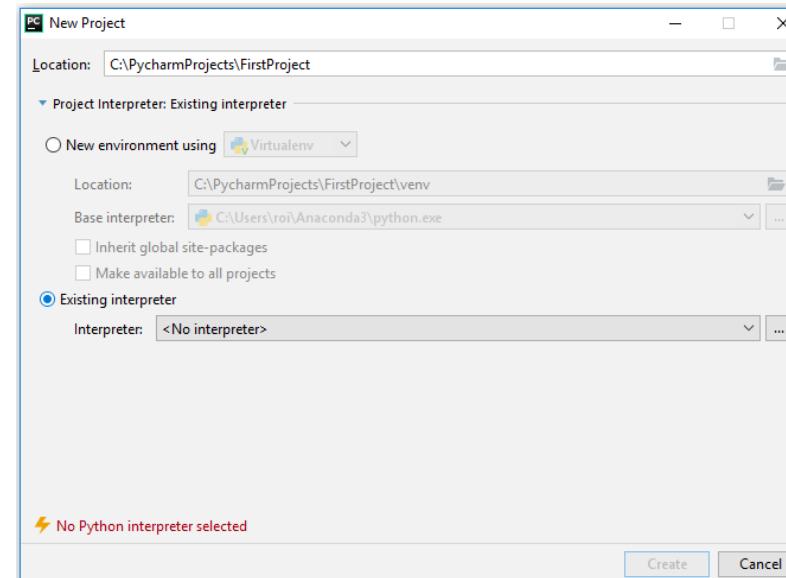
Creating New Project

- ▶ Click Create New Project



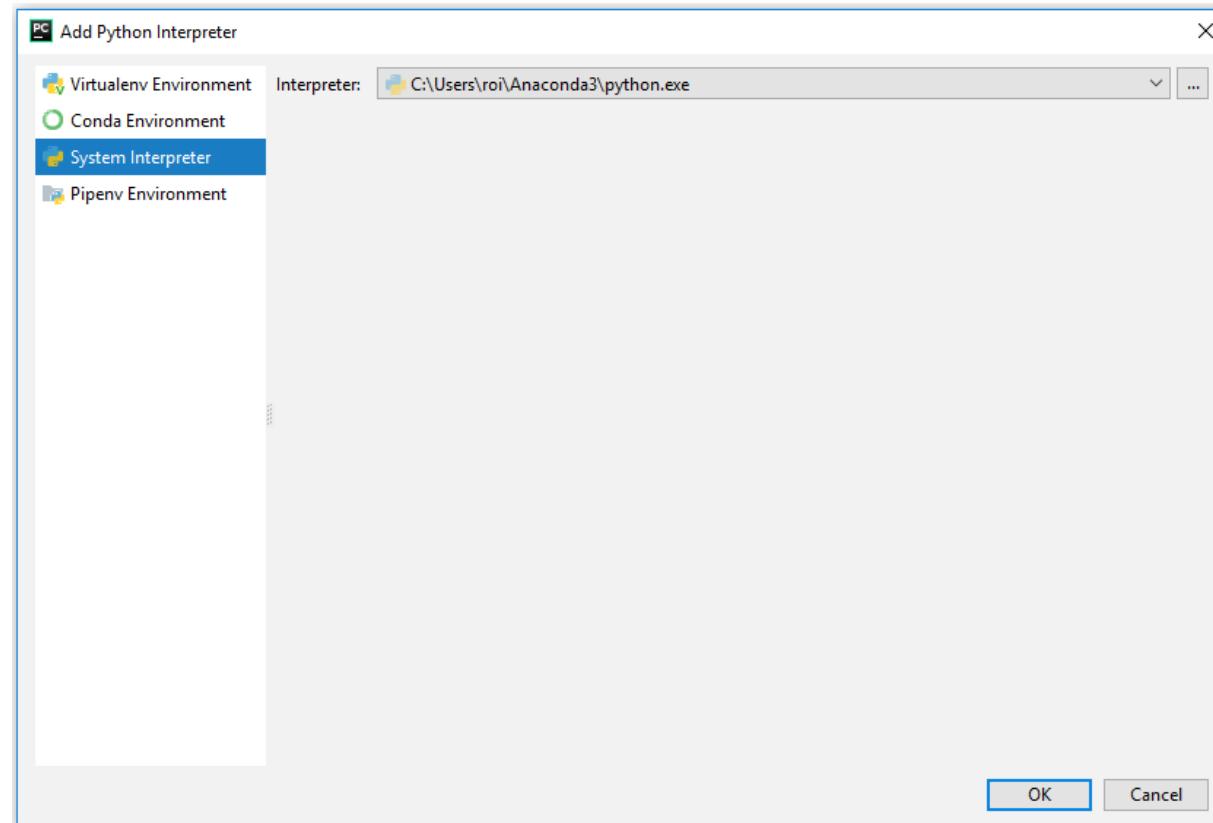
Creating New Project

- ▶ Choose a folder for your project
- ▶ Choose Existing interpreter
 - ▶ **virtualenv** is a tool to create isolated Python environments, which enables multiple side-by-side installations of Python, one for each project
- ▶ Double the ellipsis button next to the interpreter list box



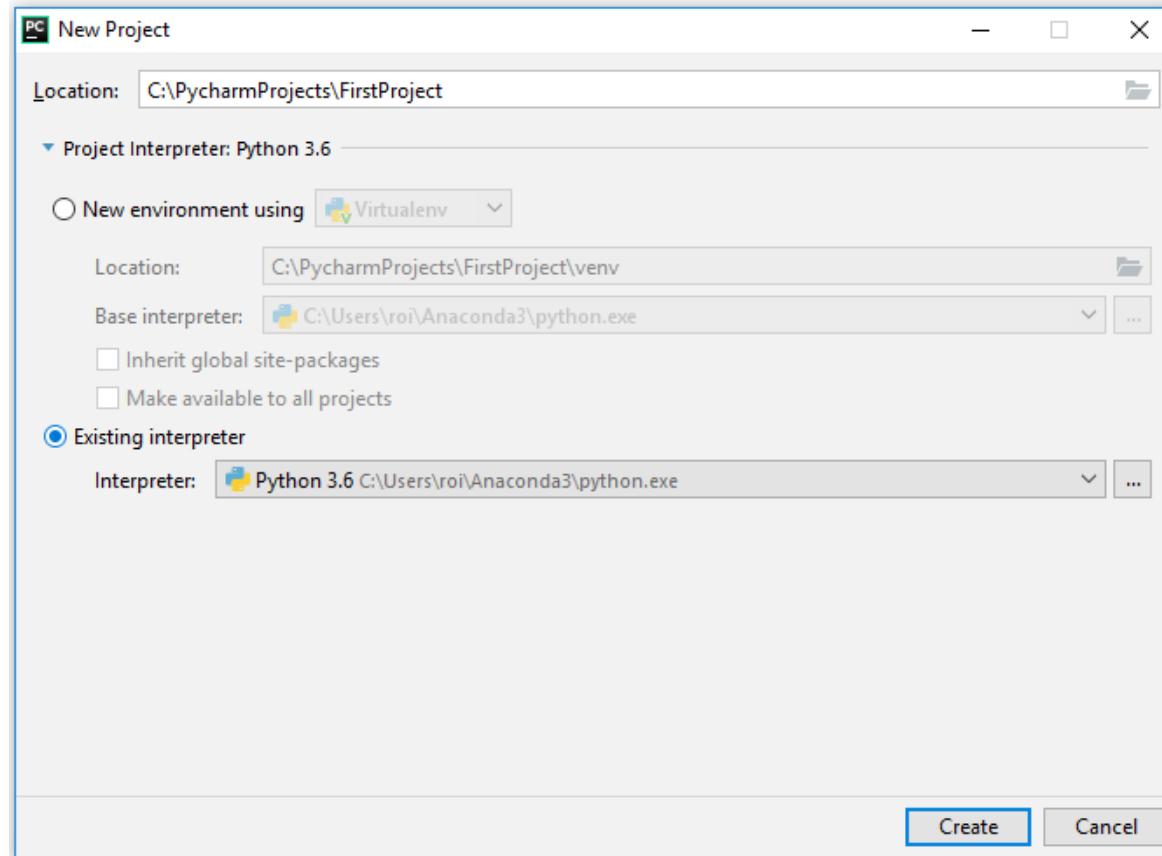
Creating New Project

- ▶ Go to System interpreter
- ▶ Choose the Anaconda python interpreter

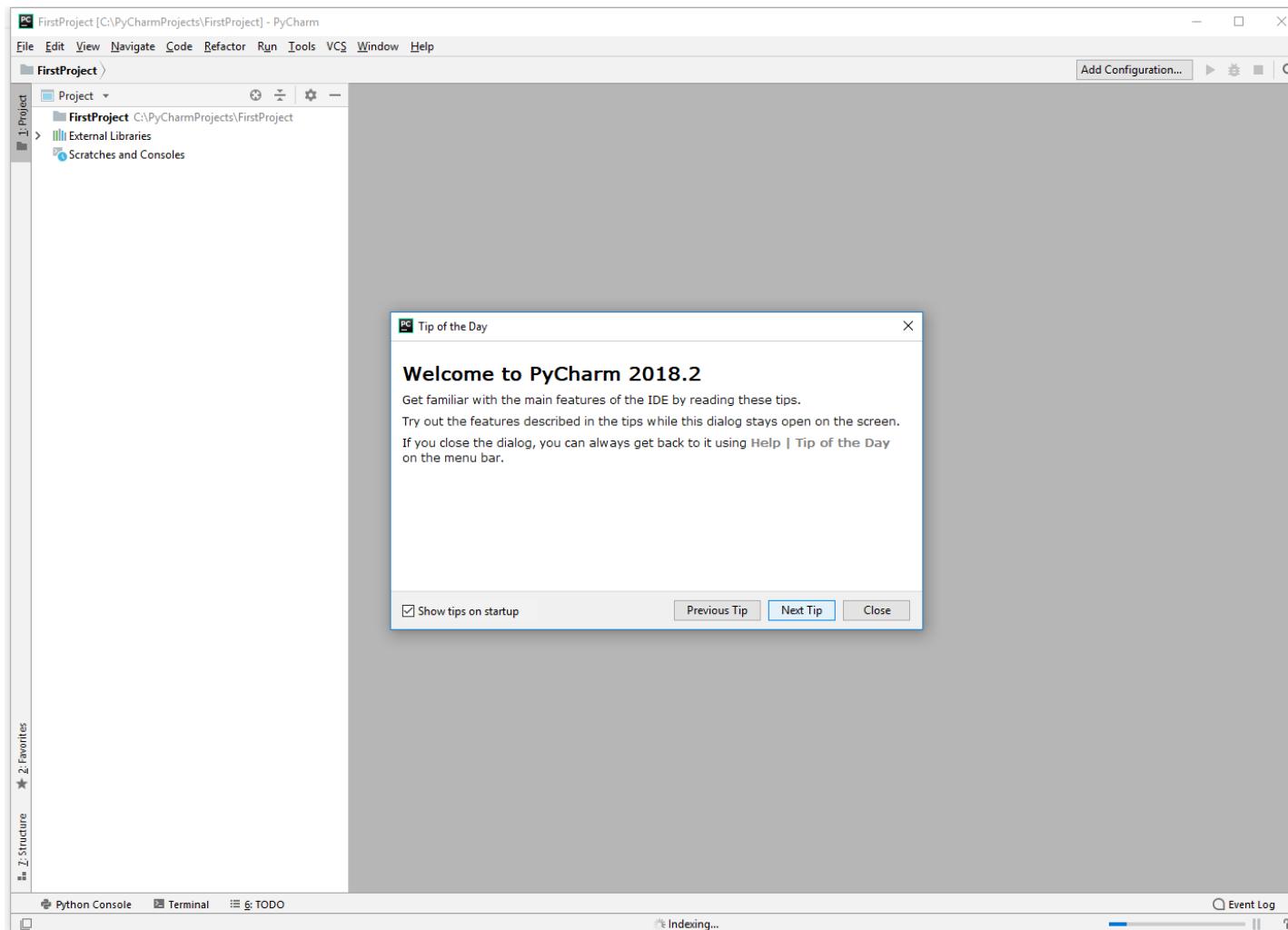


Creating New Project

- ▶ Click Create

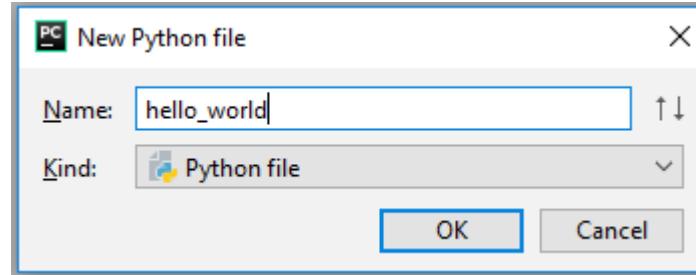


Creating New Project

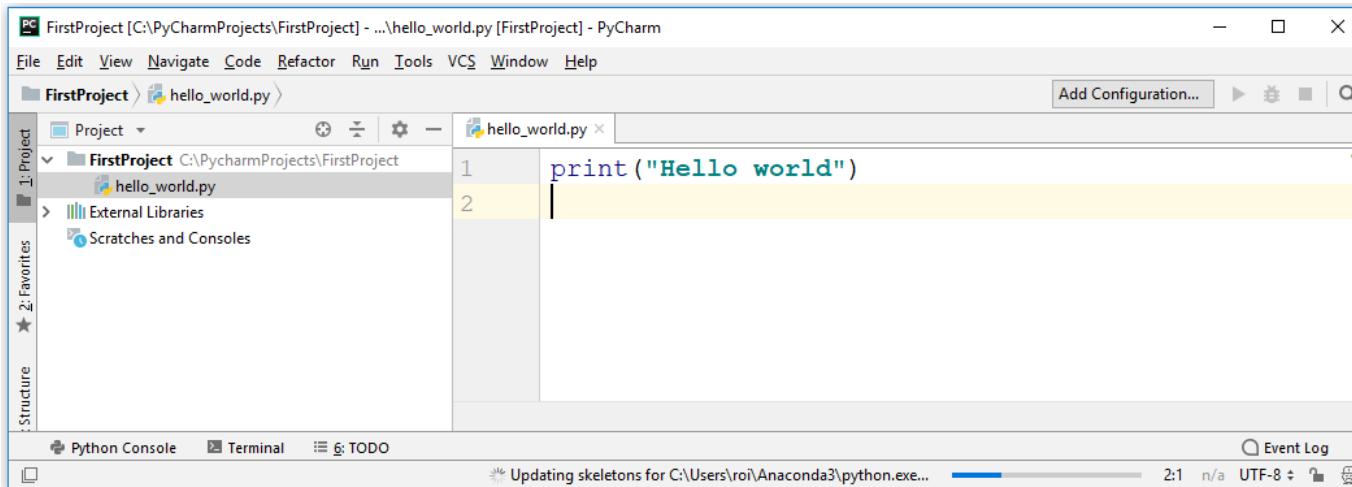


Adding New Python File

- To add a new Python file right-click the project name and choose New -> Python file



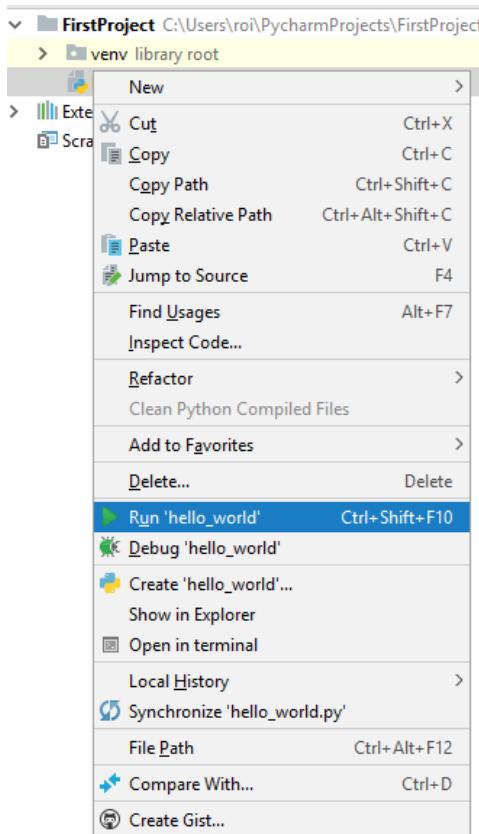
- Double-click `hello_world.py` and enter the following code:



```
1 print("Hello world")
```

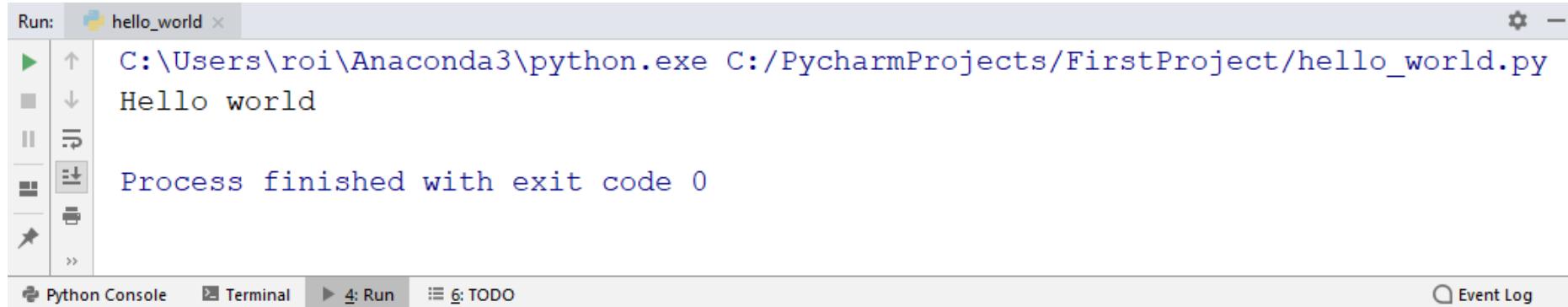
Running the Script

- ▶ Right-click the python file and choose Run



Running the Script

- ▶ The output of your script appears at the bottom of the screen:



The screenshot shows the PyCharm interface during a run session. The title bar says "Run: hello_world". The main area displays the command line output:
C:\Users\roi\Anaconda3\python.exe C:/PycharmProjects/FirstProject/hello_world.py
Hello world
Process finished with exit code 0

- ▶ If you need to run the script again, you can click the Play button on the left pane

Exercise (4)

- ▶ Create a new Pycharm project in C:\PycharmProjects named WelcomeToDS
- ▶ Add a script my_quote.py to this project
- ▶ In the script print your favorite quote to the console
- ▶ Run the script

Variables

- ▶ Variables are used to store data in your programs
- ▶ To create a variable in Python we use an assignment statement:

```
variable_name = expression
```

- ▶ `variable_name` is the name of the variable, `(=)` is the assignment operator, and `expression` is just a combination of values, variables and operators
- ▶ Example for creating a variable:

```
num = 5
```

```
num
```

```
5
```

- ▶ Python is a dynamically-typed language - we don't need to declare the type of the variable ahead of time
 - ▶ The variable's type is inferred from its definition, e.g., the number 5 is assumed to be an int

Objects

- ▶ Objects are combinations of variables, functions, and data structures
- ▶ Objects represent the entities in your program
- ▶ Everything in Python is an object (including basic types such as integers, strings, etc.)
- ▶ When an object is initiated, it is assigned a unique **object id**
- ▶ The built-in function **`id()`** returns the identity of an object as an integer
 - ▶ The id typically corresponds to the object's location in memory, but this is platform-dependent
- ▶ For example:

```
id(1.5)
```

```
2337634419600
```

Variables and Objects

- ▶ A variable name can be assigned (“bound”) to any object and used to identify that object in future calculations
- ▶ For example, when Python encounters the statement **num = 5**, it does the following:
 - ▶ Creates an int object with the value 5
 - ▶ Assign the variable name **num** to this object
 - ▶ num contains the memory location where the object 5 is stored



```
num = 5
id(num)
```

1831890528

Variables and Objects

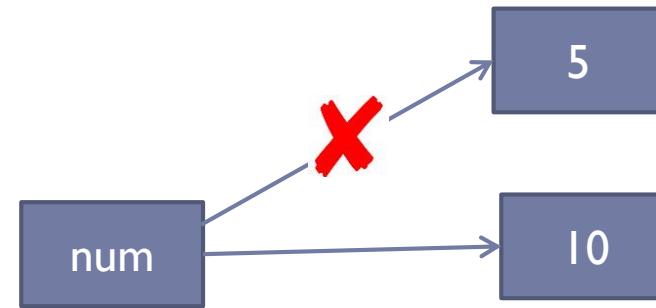
- ▶ When we assign a new value to a variable the reference to the old value is lost
- ▶ For example, when "num" is assigned to the value 10, the reference to value 5 is lost
- ▶ At this point, no variable is pointing to the memory location of the object 5
- ▶ When this happens, Python Interpreter automatically removes the object from the memory through a process known as **garbage collection**

```
num = 5  
id(num)
```

1831890528

```
num = 10  
id(num)
```

1831890688



Printing Variables

- ▶ You can print the value of a variable using the **print()** function

```
num = 12
```

```
print(num)
```

```
12
```

- ▶ Trying to access a variable before assigning any value to it results in a **NameError**:

```
age
```

```
NameError
```

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

```
<ipython-input-23-d075b0315d53> in <module>()
```

```
----> 1 age
```

```
NameError: name 'age' is not defined
```

Variable Names

- ▶ In Python, we have the following rules to create a valid variable name:
 - ▶ Only use letters (a-z, A-Z), underscore (_) and numbers (0-9)
 - ▶ It must begin with a letter or an underscore (_)
 - ▶ You can't use reserved keywords to create variables names
 - ▶ Variable name can be of any length
- ▶ Examples for valid names: home4you, after_you, _thatsall, all10, python_101
- ▶ Examples for invalid names: \$money, hello pi, 2001, break
- ▶ Python is case-sensitive language, e.g., HOME and Home are two different variables
- ▶ By convention, variable names should be lowercase, with words separated by underscores as necessary to improve readability, e.g., grades_average (another option is to use camelCase)

Python Keywords

- ▶ Python Keywords are words that have a specific meaning in the Python language
- ▶ That's why, we are not allowed to use them as variables names
- ▶ Here is the list of Python keywords:

and	assert	break	class	continue
def	del	elif	else	except
finally	for	from	global	if
import	in	is	lambda	nonlocal
not	or	pass	print	raise
return	try	while	yield	

Comments

- ▶ Comments are used to add notes to a program and help other understand your code
- ▶ In Python, everything from # to end of the line is considered a comment

```
# This is a comment on a separate line
print("Testing comments") # This is a comment after print statement
```

Testing comments

- ▶ Generally, prefer to place comments on their own lines rather than “inline” with code
- ▶ Explain *why* your code does what it does, don’t simply explain *what* it does
- ▶ A bad comment:

```
# Increase i by 10
i += 10
```

- ▶ A good comment:

```
# Skip the next 10 data points
i += 10
```

Comments

- ▶ Some programmers advocate aiming to minimize the number of comments by carefully choosing meaningful identifier names
- ▶ For example, if we rename our index, we might even do away with the comment altogether:

```
DATA_SKIP = 10  
  
data_index += DATA_SKIP
```

- ▶ Keep comments up-to-date with the code they explain
- ▶ It is all too easy to change code without synchronizing the corresponding comments:

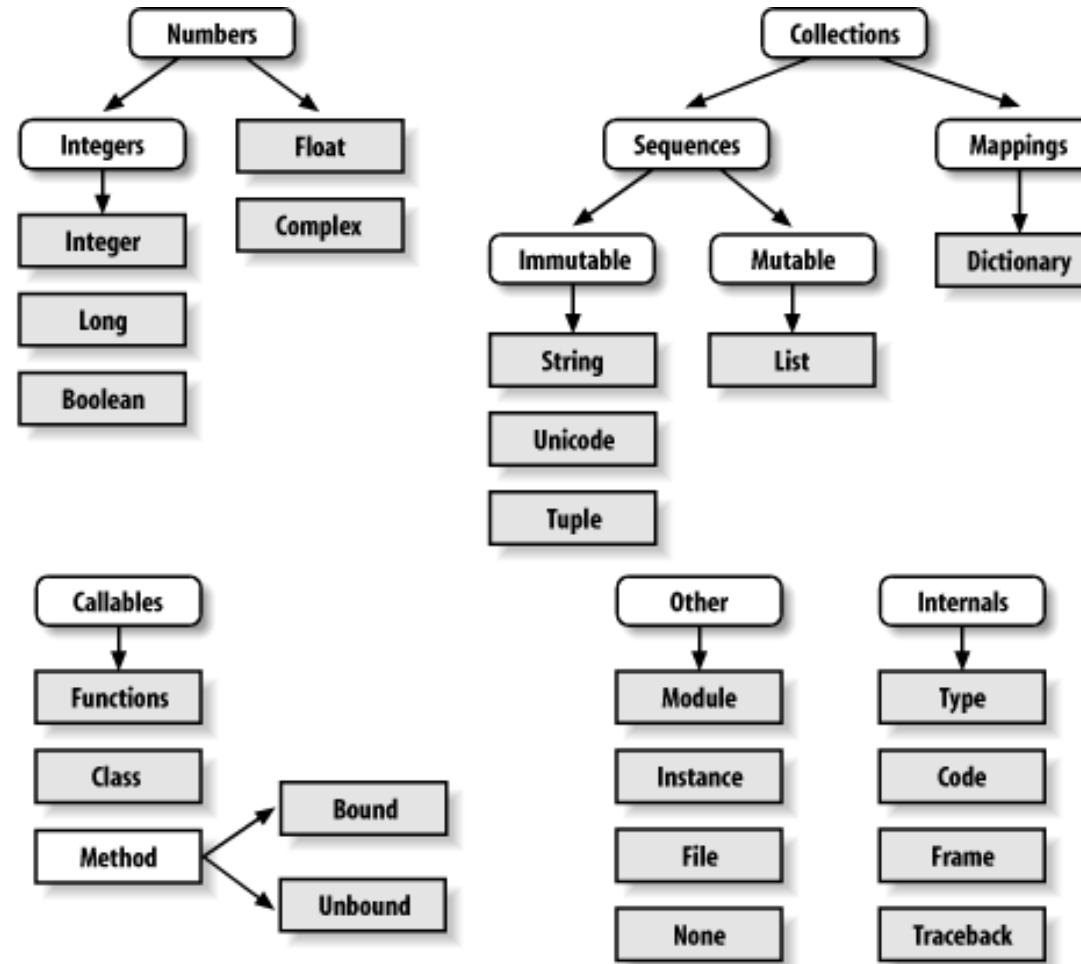
```
# Skip the next 10 data points  
i += 20
```

Data Types

- ▶ A data type defines a set of values along with operations that can be performed on those values
- ▶ Each variable and value we use in our programs has a type associated with it
- ▶ Python has 5 basic data types:

Type	Description
int	Whole numbers of unlimited range (limited only by the available memory) Older versions of Python had separate types for int and long
float	Double precision floating-point numbers, which provide approximately 15-16 digits of precision (like double in C)
complex	Complex numbers are represented as a pair of floating-point numbers The real and imaginary parts of a complex number z are available in z.real and z.imag.
bool	boolean (True or False)
str	Sequences of characters

Python Built-In Type Hierarchy



type()

- ▶ The built-in function **type()** can be used to determine the data type of the object referred by a variable:

```
num = 5
type(num)
```

int

```
s = "hello"
type(s)
```

str

```
price = 2.3
type(price)
```

float

```
allowed = True
type(allowed)
```

bool

Literals

- ▶ Explicit values we use in our programs is known as **literal**
 - ▶ For example, 10, 88.22, 'test' are called literals
- ▶ Literals also have types associated with them

```
type(54)
```

int

```
type("a string")
```

str

```
type(3.14)
```

float

```
type("3.14")
```

str

Types of Numbers

- ▶ Numbers in Python come in three types:
 - ▶ integers (type: int)
 - ▶ floating point numbers (type: float)
 - ▶ complex numbers (type: complex)
- ▶ Any single number containing a period (.) is considered by Python to specify a floating point number
- ▶ Scientific notation is supported using 'e' or 'E' to separate the significand (mantissa) from the exponent
 - ▶ For example, 1.67263e-7 represents the number 1.67263×10^{-7}
- ▶ Complex numbers such as 4+3j consist of a real and an imaginary part (denoted by j in Python), each of which is itself represented as a floating point number (even if specified without a period)

Types of Numbers

```
5
```

```
5
```

```
5.
```

```
5.0
```

```
0.100
```

```
0.1
```

```
0.0001
```

```
0.0001
```

```
# Numbers smaller than 0.0001 are displayed in scientific  
# notation  
0.00009999
```

```
9.999e-05
```

```
5.123e8
```

```
512300000.0
```

```
3j
```

```
3j
```

```
2 + 3j
```

```
(2+3j)
```

```
# A complex number may be specified by separating the real  
# and imaginary # parts in a call to complex()  
complex(2.3, 1.2)
```

```
(2.3+1.2j)
```

```
complex(5)
```

```
(5+0j)
```

Dynamic Typing

- ▶ A variable in Python can contain any data
- ▶ A variable can at one moment be a string and later receive a numeric value:

```
foo = 42      # foo is now a number
print(type(foo))

foo = 'bar'   # foo is now a string
print(type(foo))

foo = True    # foo is now a boolean
print(type(foo))

<class 'int'>
<class 'str'>
<class 'bool'>
```

- ▶ Python automatically detects the type of the variable and operations that can be performed on it based on the type of the value it contains

Dynamic Typing

- ▶ Python is a **strongly, dynamically** typed language
- ▶ **Dynamic typing** means that there are data types, but variables are not bound to any of them
- ▶ **Strong typing** means that the type of a value doesn't suddenly change
 - ▶ A string containing only digits doesn't magically become a number, as may happen in JavaScript
 - ▶ Every change of type requires an explicit conversion

Named Constants

- ▶ Constants are variables whose values don't change during the lifetime of the program
- ▶ Python doesn't have a special syntax to create constants
- ▶ We create constants just like ordinary variables
- ▶ However, to separate them from an ordinary variable, we use all uppercase letters

```
DOLLAR_TO_EURO = 0.848
DOLLAR_TO_POUND = 0.748

price = 100 * DOLLAR_TO_EURO
print(price)
```

84.8

Displaying Multiple Items with print()

- ▶ We can use the **print()** function to print multiple items in a single call by separating each item with a comma (,)
- ▶ The items will be printed to the console separated by spaces

```
age = 25
print("Your age is:", age)
```

Your age is: 25

Reading Input from Keyboard

- ▶ The function **input()** is used to read input from the keyboard. The syntax is:

```
var = input(prompt)
```

- ▶ prompt is an optional string which instructs the user to enter the input
- ▶ The **input()** function reads the input data from the keyboard and returns it as a string
- ▶ The entered data is then assigned to a variable named var for further processing

```
name = input("Enter your name: ")  
age = input("Enter your age: ")
```

Enter your name: John
Enter your age: 25

```
name
```

'John'

```
age
```

'25'

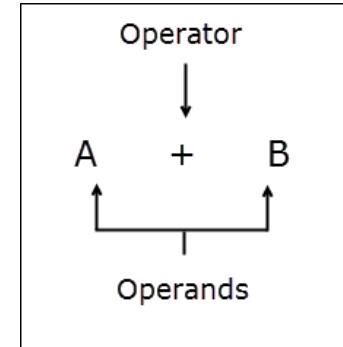
Exercise (1)

- ▶ Write a Python program which accepts the user's first and last name
- ▶ Then it prints a welcome message that starts with the word “Welcome,” followed by the user's first and last name in reverse order with a space between them
- ▶ For example:

```
Please enter your first name: Roi
Please enter your last name: Yehoshua
Welcome, Yehoshua Roi
```

Operators

- ▶ **Operator:** An operator is a symbol which specifies a specific action
- ▶ **Operand:** An operand is a data item on which operator acts



- ▶ Some operators require two operands (binary operators) while others require only one (unary operators)
- ▶ **Expression:** An expression is a combination of operators, variables, constants and function calls that results in a value
- ▶ For example:

```
1 + 8
(3 * 9) / 5
a * b + c * 3
a + b * math.pi
d + e * math.sqrt(441)
```

Arithmetic Operators

- ▶ Arithmetic operators are commonly used to perform numeric calculations
- ▶ Python has the following arithmetic operators:

Operator	Description	Example
+	Addition	$100 + 45 = 145$
-	Subtraction	$500 - 65 = 435$
*	Multiplication	$25 * 4 = 100$
/	Float division	$10 / 2 = 5.0$
//	Integer division	$10 / 2 = 5$
**	Exponentiation	$5 ** 3 = 125$
%	Remainder	$10 \% 3 = 1$

- ▶ Note that + and - operators can be binary as well as unary
 - ▶ For example: in -5 , the - operator is acting as a unary operator, whereas in $100 - 40$, it is acting as a binary operator

Float Division Operator (/)

- ▶ The / operator performs a floating point division
- ▶ Always returns a floating point number result, even if it acts on integers

```
6 / 3
```

2.0

```
2 / 3
```

0.6666666666666666

```
50 / 2.5
```

20.0

```
-5 / 2.1
```

-2.380952380952381

Integer Division Operator (//)

- ▶ The // operator always **rounds down** the result to the nearest integer
- ▶ The type of the result is int only if both operands are int, otherwise it returns a float

```
6 // 3
```

2

```
9 // 2
```

4

```
2.7 // 2
```

1.0

```
-5 // 2
```

-3

Exponentiation Operator (**)

- ▶ We use a**b operator to calculate a^b

```
15 ** 2
```

225

```
3 ** 4
```

81

```
5 ** 1.2
```

6.898648307306074

```
9 ** 0.5
```

3.0

Remainder Operator (%)

- ▶ The % operator returns the remainder after dividing left operand by the right operand
- ▶ Again the number returned is an int only if both operands are int

```
5 % 2      # 5 = (2 * 2) + 1
```

1

```
13 % 5     # 13 = (2 * 5) + 3
```

3

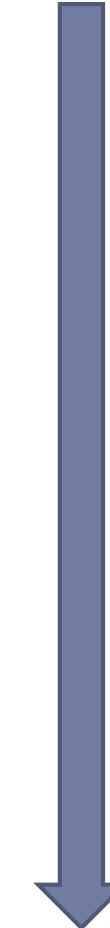
```
-13 % 5    # -13 = (-3 * 5) + 2
```

2

- ▶ The remainder operator % is a very useful operator in programming
 - ▶ One common use of % operator is to determine whether a number is even or not

Operator Precedence and Associativity

Operator	Description	Associativity
[v1, ...], { v1, ... }, { k1: v1, ... }, (...)	List/set/dict/generator creation or comprehension, parenthesized expression	left to right
seq [n], seq [n : m], func (args...), obj.attr	Indexing, slicing, function call, attribute reference	left to right
**	Exponentiation	right to left
+x, -x, ~x	Positive, negative, bitwise not	left to right
*, /, //, %	Multiplication, float division, integer division, remainder	left to right
+, -	Addition, subtraction	left to right
<<, >>	Bitwise left, right shifts	left to right
&	Bitwise and	left to right
	Bitwise or	left to right
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparision, membership and identity tests	left to right
not x	Boolean NOT	left to right
and	Boolean AND	left to right
or	Boolean OR	left to right
if-else	Conditional expression	left to right
lambda	lambda expression	left to right



Operator precedence goes from higher to lower

Operator Precedence

- ▶ Whenever we have an expression where operators involved are of different precedence, the operator with a higher precedence is evaluated first:

```
10 + 5 * 3
```

25

```
3 * 5 ** 2
```

75

- ▶ We can change the operator precedence by wrapping parentheses around the expression which we want to evaluate first:

```
(10 + 5) * 3
```

45

```
(3 * 5) ** 2
```

225

Operator Associativity

- ▶ Operator associativity defines the direction in which operators of the same precedence are evaluated
- ▶ The associativity of all arithmetic operators is from left to right
- ▶ Except for the exponentiation operator `**` which is evaluated from right to left
 - ▶ Since $a^{b^{c}} = a^{(b^c)} = a^{**}(b^{**}c)$

```
5 + 12 / 2 * 4 # 5 + ((12 / 2) * 4) == 5 + 6.0 * 4
```

29.0

```
2 ** 2 ** 3 # 2**(2**3) == 2**8
```

256

```
(2 ** 2) ** 3 # 4**3
```

64

Type Conversions

- ▶ When a calculation involves data of different types Python has the following rules:
 - ▶ When both operands are int, the result will be an int
 - ▶ When both operands are float, the result will be a float
 - ▶ When one operand is of float type and the other is of type int then the result will be a float
- ▶ Sometimes, we need to convert data from one type to a different type
 - ▶ For example, when reading numeric inputs from the console
- ▶ Python provides us the following conversion functions:

Function	Description	Examples
int()	Accepts a string or a number and returns a value of type int. A floating point number is rounded down in casting it into an int.	int(2.7) returns 2 int("25") returns 25
float()	Accepts a string or a number and returns a value of type float	float(42) return 42.0 float("1.6") returns 1.6
str()	Accepts any value and returns a value of type str	str(12) returns "12" str(3.4) returns "3.4"

Type Conversions

- ▶ A program that asks the user to enter two numbers and prints their sum:

```
num1 = int(input("Enter first number: ")) # int() is used to convert the input to a number
num2 = int(input("Enter second number: "))

sum = num1 + num2
print("Their sum is:", sum)
print("Their sum is: " + str(sum)) # another way to print a string together with a number
```

```
Enter first number: 5
Enter second number: 7
Their sum is: 12
Their sum is: 12
```

Exercise (2)

- Predict the results of the following expressions and check them in Jupyter Notebook:

```
2.7 / 2
2 / 4 - 1
2 // 4 - 1
(2 + 5) % 3
2 + 5 % 3
3 * 4 // 6
3 * (4 // 6)
3 * 2 ** 2
3 ** 2 * 2
-2 ** 2
2 ** -2
(-2) ** 2
-2 ** 3 ** 2
(-2) ** 3 ** 2
```

Exercise (3)

- ▶ Write a program that asks the user to enter the length and width of a rectangle, and prints its area and perimeter

```
length = int(input("Enter the length of the rectangle: "))
width = int(input("Enter the width of the rectangle: "))

area = length * width
perimeter = 2 * (length + width)

print("The rectangle's area is:", area)
print("The rectangle's perimeter is:", perimeter)
```

```
Enter the length of the rectangle: 5
Enter the width of the rectangle: 7
The rectangle's area is: 35
The rectangle's perimeter is: 24
```

Breaking Statements into Multiple Lines

- ▶ Python doesn't place any restriction on the length of a line,
- ▶ However, for ease of reading, it is usually a good idea to keep the lines of your program to a fixed maximum length (80 characters is recommended)
- ▶ There are two ways in Python to break long statements into multiple lines:
- ▶ Using parenthesis:

```
(1111100 + 45 - (88 / 43) + 783 +
 10 - 33 * 1000 +
 88 + 3772)
```

1082795.953488372

- ▶ Using the line continuation symbol (\):

```
1111100 + 45 - (88 / 43) + 783 + \
 10 - 33 * 1000 +
 88 + 3772
```

1082795.953488372

Compound Assignment Operator

- ▶ In programming it is very common to change the value of a variable and then reassign the value back to the same variable, i.e., $a = a + 5$
- ▶ Such reassessments have useful shorthand notation: the **compound assignment operator**
- ▶ The following table lists the compound assignment operators available in Python:

Operator	Example	Equivalent to
<code>+=</code>	<code>x += 5</code>	<code>x = x + 5</code>
<code>-=</code>	<code>x -= 5</code>	<code>x = x - 5</code>
<code>*=</code>	<code>x *= 5</code>	<code>x = x * 5</code>
<code>/=</code>	<code>x /= 5</code>	<code>x = x / 5</code>
<code>//=</code>	<code>x //= 5</code>	<code>x = x // 5</code>
<code>%=</code>	<code>x %= 5</code>	<code>x = x % 5</code>
<code>**=</code>	<code>x **= 5</code>	<code>x = x ** 5</code>

Compound Assignment Operators

▶ Example:

```
x = 5
x *= 2
x
```

10

```
x += 1
x
```

11

- ▶ C-style increment and decrement operations such as `a++` for `a += 1` are *not supported* in Python

Exercise (4)

- ▶ Write a program that asks the user to enter a 3-digit number and prints the sum of its digits
- ▶ Use the compound assignment operators
- ▶ Sample run:

```
Enter a 3-digit number: 582
Sum of digits is: 15
```

Methods and Attributes of Numbers

- ▶ Python numbers are **objects** (everything in Python is an object)
- ▶ Thus, they have **attributes**, accessed using the “dot” notation `<object>.<attribute>`
- ▶ For example, complex number objects have the attributes **real** and **imag**, which are the real and imaginary (floating point) parts of the number:

```
(4+5j).real
```

4.0

```
(4+5j).imag
```

5.0

- ▶ Numbers also have **methods**: functions that belong to objects
- ▶ Methods are called on an object using the following notation:
`object.method_name(arg1, arg2, ..., argN)`

Methods and Attributes of Numbers

- ▶ For example, complex numbers have a method, **conjugate()**, which returns the complex conjugate:

```
(4+5j).conjugate()
```

```
(4-5j)
```

- ▶ Integers and floating point numbers don't actually have many useful attributes
- ▶ But if you're curious you can find out how many bits an integer takes up in memory by calling its **bit_length()** method:

```
(5733382412312391233243).bit_length()
```

```
73
```

Mathematical Functions

- ▶ Python provides the following built-in mathematical functions:

Function	Description	Example
<code>abs(number)</code>	Returns the absolute value of <i>number</i>	<code>abs(-12) = 12</code> <code>abs(112.21) = 112.21</code>
<code>round(number)</code>	Rounds <i>number</i> to the nearest integer	<code>round(17.3) = 17</code> <code>round(8.6) = 9</code>
<code>round(number, ndigits)</code>	Rounds <i>number</i> to <i>ndigits</i> after decimal point	<code>round(3.14159, 2) = 3.14</code> <code>round(2.71828, 2) = 2.72</code>
<code>min(arg1, arg2, ..., argN)</code>	Returns the smallest item among <i>arg1, arg2, ..., argN</i>	<code>min(12, 2, 44, 199) = 2</code>
<code>max(arg1, arg2, ..., argN)</code>	Returns the largest item among <i>arg1, arg2, ..., argN</i>	<code>max(991, 22, 19) = 991</code>

Mathematical Functions

- ▶ Examples for the `abs()` function:

```
abs(-5.2)
```

5.2

```
abs(-2)
```

2

```
abs(3+4j)
```

5.0

- ▶ This is an example of *polymorphism*: the same function, `abs`, does different things to different objects:
 - ▶ If passed a real number, x , it returns $|x|$, the non-negative magnitude of that number
 - ▶ If passed a complex number, $z = x + iy$, it returns the modulus $z = \sqrt{x^2 + y^2}$

Mathematical Functions

- ▶ Examples for the **round()** function:

```
round(-9.62)
```

-10

```
round(7.5)
```

8

```
round(4.5)
```

4

- ▶ Note that in Python 3, this function employs *Banker's rounding*: if a number is midway between two integers, then the even integer is returned

Modules in Python

- ▶ Python uses modules to group related functions, classes, and variables
 - ▶ For example, the **math** module contains various mathematical functions
 - ▶ **datetime** module contains various classes and functions for working with dates and times
 - ▶ To use functions, constants or classes defined inside a module, we first have to **import** it using the import statement
 - ▶ This statement finds and loads the module into memory
 - ▶ The syntax of the import statement is as follows:
- ```
import module_name
```
- ▶ For example, to import the math module in a Jupyter Notebook write:

```
import math
```

# Modules in Python

- ▶ To use a method or a constant from a module, you type the module name followed by the dot (.) operator, and the name of the method or constant that you need
- ▶ For example, to use the **sqrt()** function from the math module, type:

```
math.sqrt(225)
```

```
15.0
```

- ▶ It is possible to import the math module with ‘from math import \*’ and access its functions directly:

```
from math import *
cos(pi)
```

```
-1.0
```

- ▶ However, this is not recommended in Python programs, since there is a danger of name conflicts (particularly if many modules are imported in this way)

# The math Module (1)

- ▶ Some standard functions and constants provided by the math module:

| Function/Constant | Description                                             | Example                   |
|-------------------|---------------------------------------------------------|---------------------------|
| math.pi           | The value of $\pi$                                      |                           |
| math.e            | The value of e                                          |                           |
| math.ceil(x)      | Returns the smallest integer greater than or equal to x | math.ceil(3.621) = 4      |
| math.floor(x)     | Returns the largest integer smaller than or equal to x  | math.floor(3.621) = 3     |
| math.sqrt(x)      | Returns the square root of x as float                   | math.sqrt(144) = 12.0     |
| math.exp(x)       | Returns the exponent of x ( $e^x$ )                     | math.exp(2) = 7.3891      |
| math.log(x)       | Returns the natural log of x to the base e              | math.log(2) = 0.6931      |
| math.log10(x)     | Returns the log of x to the base 10                     | math.log10(999) = 2.9996  |
| math.log(x, b)    | Returns the log of x to the given base b                | math.log(2, 2) = 1.0      |
| math.sin(x)       | Returns the sine of x radians                           | math.sin(math.pi/2) = 1.0 |
| math.cos(x)       | Returns the cosine of x radians                         | math.cos(0) = 1.0         |
| math.tan(x)       | Returns the tangent of x radians                        | math.tan(45) = 1.61       |

# The math Module (2)

- ▶ More math functions:

| Function/Constant | Description                                | Example                      |
|-------------------|--------------------------------------------|------------------------------|
| math.degrees(x)   | Converts the angle from radians to degrees | math.degrees(math.pi/2) = 90 |
| math.radians(x)   | Converts the angle from degrees to radians | math.radians(90) = 1.5707    |
| math.hypot(x, y)  | The Eculidean norm $\sqrt{x^2 + y^2}$      | math.hypot(3,5) = 5.38095    |
| math.factorial(x) | $x!$                                       | math.factorial(5) = 120      |

- ▶ A complete list of the functions and constants provided by the math module can be found at <https://docs.python.org/3/library/math.html>

# Math Functions – Example

```
import math
```

```
math.ceil(3.5) # the smallest integer greater than or equal to 3.5
```

  
4

```
math.floor(3.5) # the largest integer smaller than or equal to 3.5
```

  
3

```
math.sqrt(144) # square root of 144
```

  
12.0

```
math.log(2) # find Log of 2 to the base e
```

  
0.6931471805599453

```
math.log(2, 5) # find log of 2 to the base 5
```

  
0.43067655807339306

```
math.cos(0)
```

  
1.0

```
math.cos(0)
```

  
1.0

```
math.sin(math.pi/2)
```

  
1.0

```
math.degrees(math.pi/2)
```

  
90.0

```
math.hypot(3,5) # The Eculidean norm sqrt(3**2 + 5**2)
```

  
5.830951894845301

```
math.factorial(5) # 5! = 1 * 2 * 3 * 4 * 5
```

  
120

## Exercise (5)

- ▶ Write a program that computes the area of a triangle given its sides
- ▶ Get from the user the lengths of the three triangle sides:  $a, b, c$
- ▶ Use Heron's formula to compute the triangle's area:

$$A = \sqrt{s(s-a)(s-b)(s-c)}, \quad \text{where } s = \frac{1}{2}(a+b+c)$$

- ▶ For example, if  $a = 4.503, b = 2.377, c = 3.902$ , the area is 4.63511

## Exercise (6)

- ▶ Explain the (surprising?) behavior of the following short code:

```
d = 8
e = 2
from math import *
sqrt(d ** e)
```

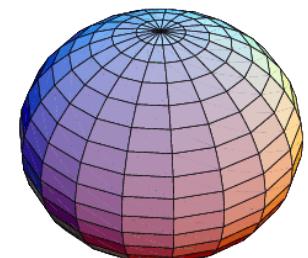
16.88210319127114

## Exercise (7)

- ▶ The Earth's surface area can be approximated to the form of an oblate spheroid with semi-major and semi-minor axes  $a = 6378137.0$  m and  $c = 6356752.314245$  m
  - ▶ Oblate shperiod is a sphere that is wider at its horizontal axis than it is at its vertical axis
- ▶ Use the formula for the surface area of an oblate spheroid to calculate the surface area of Earth:

$$S_{obl} = 2\pi a^2 \left( 1 + \frac{1-e^2}{e} \operatorname{atanh}(e) \right), \quad \text{where } e^2 = 1 - \frac{c^2}{a^2}$$

- ▶ Compare it with the surface area of Earth if it were assumed to be a perfect sphere with radius 6371 km  $S_{sphere} = 4\pi r^2$



## Exercise (8)

---

- ▶ Get a positive integer  $x$  from the user
- ▶ Print the number of digits that the number  $x$  has
  - ▶ For example, if  $x = 5731$  you should print 4
- ▶ Use only numerical functions and operators
  - ▶ i.e., you're not allowed to convert the number into a string

# The bool Type

- ▶ The **bool** data type represents two states: true or false
- ▶ The reserved keywords **True** and **False** define the values true and false
- ▶ A variable of type **bool** can contain only one of these two values

```
var1 = True
```

```
type(var1)
```

```
bool
```

```
var1
```

```
True
```

- ▶ Internally, Python uses 1 and 0 to represent True and False respectively:

```
int(True)
```

```
1
```

```
int(False)
```

```
0
```

# Truthy and Falsy Values

- ▶ **Truthy values:** Values which are equivalent to the bool value True
- ▶ **Falsy values:** Values which are equivalent to the bool value False
- ▶ In Python, the following values are considered falsy:
  - ▶ False
  - ▶ Zero: 0, 0.0
  - ▶ None
  - ▶ Empty sequence, e.g., "", [], ()
  - ▶ Empty dictionary {}
- ▶ Everything else is considered as truthy
- ▶ We can also test whether a value is truthy or falsy by using the **bool()** function
  - ▶ If value a truthy then `bool()` function returns True, otherwise it returns False

# Truthy and Falsy Values

```
bool("") # an empty string is a falsy value
```

False

```
bool("bool") # the string "bool" is a truthy value
```

True

```
bool(12) # int 12 is a truthy value
```

True

```
bool(0) # int 0 is a falsy value
```

False

```
bool([]) # an empty list is a falsy value
```

False

```
bool(None) # None is a falsy value
```

False

# Relational Operators

- ▶ Relational operators (aka comparison operators) allow us to compare values
- ▶ If the result of the comparison is true, then a bool value True is returned, otherwise the bool value False is returned
- ▶ The relational operators available in Python:

| Operator | Description              | Example               |
|----------|--------------------------|-----------------------|
| <        | Smaller than             | $3 < 4$ (True)        |
| >        | Greater than             | $90 > 450$ (False)    |
| $\leq$   | Smaller than or equal to | $10 \leq 11$ (True)   |
| $\geq$   | Greater than or equal to | $31 \geq 40$ (False)  |
| $\neq$   | Not equal to             | $100 \neq 101$ (True) |
| $\equiv$ | Equal to                 | $50 \equiv 50$ (True) |

- ▶ Note the difference between  $\equiv$  and  $=$   
The single equals sign  $=$  is an assignment operator, used to assign a value to a variable, while the double equals sign  $\equiv$  is an equality operator, used to test whether two values are equal or not.

# Relational Operators

- ▶ Python can, as far as possible without ambiguity, compare objects of different types:

```
4 >= 3.14
```

True

```
0 == False
```

True

- ▶ Care should be taken in comparing floating point numbers for equality

- ▶ Because they are not stored exactly and calculations involving them frequently leads to a loss of precision, this can give unexpected results to the unwary:

```
a = 0.01
b = 0.1 ** 2
a == b
```

False

```
print("a =", a, "b =", b)
```

a = 0.01 b = 0.01000000000000002

# Logical Operators

- ▶ Logical operators are used to combine two or more boolean expressions and tests whether they are true or false
- ▶ Expressions containing logical operators are known as **boolean expressions** (or logical expressions)
- ▶ The logical operators available in Python:

| Operator | Description  |
|----------|--------------|
| and      | AND operator |
| or       | OR operator  |
| not      | NOT operator |

# Logical Operators

- The **and** operator returns True if both operands are true, otherwise it returns False

| P     | Q     | P and Q |
|-------|-------|---------|
| False | False | False   |
| False | True  | False   |
| True  | False | False   |
| True  | True  | True    |

- The **or** operator returns True if one of the operands is true, otherwise it returns False

| P     | Q     | P or Q |
|-------|-------|--------|
| False | False | False  |
| False | True  | True   |
| True  | False | True   |
| True  | True  | True   |

- The **not** operator negates the value of the expression

| P     | not P |
|-------|-------|
| False | True  |
| True  | False |

# Logical Operators

- ▶ In compound expressions, the comparison operators are evaluated first, and then the logic operators in order of precedence: not, and, or
- ▶ This precedence can be overridden with parentheses, as for arithmetic

```
7.0 > 4 and -1 >= 0 # equivalent to True and False
```

False

```
5 < 4 or 1 != 2 # equivalent to False or True
```

True

```
not 7.5 < 0.9 or 4 == 4
```

True

```
not (7.5 < 0.9 or 4 == 4)
```

False

# Logical Operators

- ▶ In a logic test expression, it is not always necessary to make an explicit comparison to obtain a boolean value
- ▶ Python will try to convert an object to a bool type if needed

```
a = 0
a or 4 < 3 # same as: False or 4 < 3
```

False

```
not a + 1 # same as: not True
```

False

- ▶ In the last example, addition has higher precedence than the operator not, so a+1 is evaluated first to give 1, which corresponds to boolean True, so the whole expression equals to not True
- ▶ To explicitly convert an object to a boolean object, use the bool() function:

```
bool(-1)
```

True

# Short Circuiting

- ▶ The **and** and **or** operators always return one of their operands and not just its bool equivalent:

```
a = 0
a+2 or 4 == 4
```

2

- ▶ Logic expressions are evaluated left to right, and those involving **and** or **or** are *short-circuited*: the second expression is only evaluated if necessary to decide the truth value of the whole expression

```
4 > 3 and a-2
```

-2

- ▶  $4 > 3$  is True, so the second expression must be evaluated to establish the truth of the and condition.  $a-2$  is equal to  $-2$ , which is also equivalent to True, so the and condition is fulfilled and  $-2$  (as the most recently evaluated expression) is returned.

# Short Circuiting

- ▶ A common Python idiom is to assign a variable using the return value of a logic expression:

```
a = 0
b = a or 5
b
```

5

## Exercise (9)

- Predict the results of the following expressions and check them in Jupyter Notebook:

```
1 < 2 or 4 < 2
not 1 < 2 or 4 < 2
not (1 < 2 or 4 < 2)
4 > 2 or 10/0 == 0
not 0 < 2
(not 0) < 2
1 and 2
0 and 1
1 or 0
```

## Exercise (10)

- ▶ There is no XOR (exclusive-or) operator provided “out of the box” by Python, but one can be constructed from the existing operators
- ▶ Devise two different ways of doing this
- ▶ The truth table for the xor operator is:

| P     | Q     | P xor Q |
|-------|-------|---------|
| False | False | False   |
| False | True  | True    |
| True  | False | True    |
| True  | True  | False   |

# Identity Operator

- ▶ The **is** operator is used to check if two variables refer to the same object:

```
a = 1234
b = a
a is b
```

True

```
c = 1234
c is a
```

False

```
c == a
```

True

- ▶ Here, the assignment `c = 1234` creates an entirely new integer object, so `c is a` evaluates as False, even though `a` and `c` have the same value

# Identity Operator

- ▶ Python also provides the operator **is not**, which returns true if the two variables don't refer to the same object
- ▶ It is more natural to use b is not a, than not b is a

```
a = 8
b = a
b is a
```

True

```
b /= 2
b is not a
```

True

# Identity Operator

- Given the previous discussion, the following result might come out as a surprise:

```
a = 250
b = 250
a is b
```

True

- This happens because Python keeps a **cache** of commonly used, small integer objects in order to improve performance
  - typically, the numbers –5–256
- The assignment a = 250 attaches the variable name a to the existing integer object without having to allocate new memory for it
- Because the same thing happens with b, the two variables point to the same object

# Identity Operator

- ▶ The identity operator in combination with the type() built-in function can be used to check the type of an object:

```
x = 5.2
type(x) is float
```

True

```
type(x) is int
```

False

# Special Value None

- ▶ Python defines a single value, **None**, of the special type, **NoneType**
- ▶ It is used to represent the absence of a defined value
  - ▶ For example, where no value is possible or relevant
  - ▶ This is particularly helpful in avoiding arbitrary default values for bad or missing data
- ▶ Comparisons to singletons like None should always be done with is or is not, and not with the equality operators
  - ▶ is always compares by object identity, while the result of == depends on the type of the operands

```
my_var = None
my_var is None
```

True

## Exercise (11)

- Predict the results of the following expressions and check them in Jupyter Notebook:

```
a, b = 5, 5
a == b
```

```
a is b
```

```
a * 100 == b * 100
```

```
a * 100 is b * 100
```

```
a + 0.0 is a
```

```
a = None
b = None
a == b
```

```
a is b
```

```
a > b
```

# Strings

- ▶ A string object (of type str) is an ordered, immutable sequence of characters
- ▶ To define a variable containing some constant text (a *string literal*), enclose the text in either single or double quotes:

```
str1 = 'String in single quotes'
str1
```

```
'String in single quotes'
```

```
str2 = "String in double quotes"
str2
```

```
'String in double quotes'
```

- ▶ Inside the Python Shell a string is always displayed using single quotation marks
- ▶ However, if you use the print() function only contents of the string is displayed:

```
print(str1)
print(str2)
```

```
String in single quotes
String in double quotes
```

# Strings

- ▶ Double quotes comes in handy when you have single quotation marks inside a string
- ▶ For example:

```
print("I'm learning Python")
```

```
I'm learning Python
```

- ▶ If we had used the single quotes, we would get a SyntaxError:

```
print('I'm learning Python')
```

```
File "<ipython-input-11-45cd6c952520>", line 1
 print('I'm learning Python')
 ^
SyntaxError: invalid syntax
```

- ▶ Similarly, If you want to print double quotes inside a string, just wrap the entire string inside single quotes instead of double quotes

# Strings

- ▶ Some languages like C, C++, Java treats a single character as a special type called `char`, but in Python a single character is also a string:

```
ch = 'a' # a string containing a single character
type(ch)
```

str

```
type("a string") # a string containing multiple characters
```

str

# String Concatenation

- ▶ Strings can be concatenated using either the + operator or by placing them next to each other on the same line:

```
"abc" + "def"
```

```
'abcdef'
```

```
'one ' 'two ' 'three'
```

```
'one two three'
```

- ▶ What would happen if one of the operand is not a string?

```
s = "Python" + 101
```

```

TypeError: must be str, not int
ast)
<ipython-input-19-cd2000b866bc> in <module>()
----> 1 s = "Python" + 101
```

- ▶ Python is a strongly typed language, thus it doesn't convert data of one type to a different type automatically

# String Repetition Operator (\*)

- When used with strings the \* operator repeats the string  $n$  number of times

```
5 * 'a'
```

```
'aaaaaa'
```

```
"-0-" * 5
```

```
'-0--0--0--0--0-'
```

```
print("We have got some", "spam" * 5)
```

```
We have got some spamspamspamspamspam
```

- Strings concatenated with the '+' operator can be repeated with '\*', but only if enclosed in parentheses:

```
('a' * 4 + 'B')*3
```

```
'aaaaBaaaaBaaaaB'
```

# Long Strings

- To break up a long string over two or more lines of code, use the line continuation character, '\ or (better) enclose the string literal in parentheses:

```
long_str = 'We hold these truths to be self-evident,\n ' that all men are created equal...'
long_str
```

```
'We hold these truths to be self-evident, that all men are
created equal...'
```

```
long_str = ('We hold these truths to be self-evident,'
 ' that all men are created equal...')
long_str
```

```
'We hold these truths to be self-evident, that all men are
created equal...'
```

# Escape Sequences

- ▶ Escape sequences are a set of special characters used to print characters which can't be typed directly using the keyboard
- ▶ Each escape sequence starts with a backslash ( \ ) character
- ▶ Common Python escape sequences:

| Escape Sequence | Meaning                        |
|-----------------|--------------------------------|
| \n              | Line Feed (LF)                 |
| \r              | Carriage return (CR)           |
| \t              | Horizontal tab                 |
| \b              | Backspace                      |
| \'              | Single quote                   |
| \"              | Double quote                   |
| \\"             | The backslash character itself |
| \u, \U, \N      | Unicode character              |
| \x              | Hex-encoded byte               |

# Escape Sequences

- ▶ For example, \t inside a string prints a tab character (four spaces)

```
s = "Name\tAge\tGrades"
s
```

```
'Name\tAge\tGrades'
```

```
print(s)
```

```
Name Age Grades
```

- ▶ Note that just typing a variable's name at the Python shell prompt simply echoes its literal value back to you
- ▶ \n character inside the string prints a newline character

```
s = "One\nTwo\nThree"
print(s)
```

```
One
Two
Three
```

# Escape Sequences

- ▶ You can also use the \' and \" escape sequences to print a single or a double quotation marks in a string

```
print('I\'m learning Python')
```

```
I'm learning Python
```

```
print("John says \"Hello there\"")
```

```
John says "Hello there"
```

- ▶ Similarly to print a single backslash character \ you need to double it \\

```
path = "C:\\\\Users\\\\Roi"
print(path)
```

```
C:\\\\Users\\\\Roi
```

# Raw Strings

- If you want to define a string to include character sequences such as '\n' without them being escaped, define a **raw string** prefixed with r:

```
rawstring = r'The escape sequence for a new line is \n'
rawstring
```

```
'The escape sequence for a new line is \\n'
```

```
print(rawstring)
```

```
The escape sequence for a new line is \n
```

# Triple-Quoted Strings

- ▶ When defining a block of text including several line endings it is often inconvenient to use \n repeatedly
- ▶ This can be avoided by' using **triple-quoted strings**: newlines defined within strings delimited by """ and "" are preserved in the string

```
a = """one
two
three"""
print(a)
```

```
one
two
three
```

# ASCII

- ▶ **ASCII** - The American Standard Code for Information Interchange is a standard seven-bit code that consists of 128 decimal numbers assigned to letters, numbers, punctuation marks, and the most common special characters
- ▶ **ASCII code** is the numerical representation of a character

| Non-Printable Characters |     |                           |     |     |                                 |
|--------------------------|-----|---------------------------|-----|-----|---------------------------------|
| DEC                      | HEX | CHARACTER (CODE)          | DEC | HEX | CHARACTER (CODE)                |
| 0                        | 0   | NULL                      | 16  | 10  | DATA LINK ESCAPE (DLE)          |
| 1                        | 1   | START OF HEADING (SOH)    | 17  | 11  | DEVICE CONTROL 1 (DC1)          |
| 2                        | 2   | START OF TEXT (STX)       | 18  | 12  | DEVICE CONTROL 2 (DC2)          |
| 3                        | 3   | END OF TEXT (ETX)         | 19  | 13  | DEVICE CONTROL 3 (DC3)          |
| 4                        | 4   | END OF TRANSMISSION (EOT) | 20  | 14  | DEVICE CONTROL 4 (DC4)          |
| 5                        | 5   | END OF QUERY (ENQ)        | 21  | 15  | NEGATIVE ACKNOWLEDGEMENT (NAK)  |
| 6                        | 6   | ACKNOWLEDGE (ACK)         | 22  | 16  | SYNCHRONIZE (SYN)               |
| 7                        | 7   | BEEP (BEL)                | 23  | 17  | END OF TRANSMISSION BLOCK (ETB) |
| 8                        | 8   | BACKSPACE (BS)            | 24  | 18  | CANCEL (CAN)                    |
| 9                        | 9   | HORIZONTAL TAB (HT)       | 25  | 19  | END OF MEDIUM (EM)              |
| 10                       | A   | LINE FEED (LF)            | 26  | 1A  | SUBSTITUTE (SUB)                |
| 11                       | B   | VERTICAL TAB (VT)         | 27  | 1B  | ESCAPE (ESC)                    |
| 12                       | C   | FF (FORM FEED)            | 28  | 1C  | FILE SEPARATOR (FS) RIGHT ARROW |
| 13                       | D   | CR (CARRIAGE RETURN)      | 29  | 1D  | GROUP SEPARATOR (GS) LEFT ARROW |
| 14                       | E   | SO (SHIFT OUT)            | 30  | 1E  | RECORD SEPARATOR (RS) UP ARROW  |
| 15                       | F   | SI (SHIFT IN)             | 31  | 1F  | UNIT SEPARATOR (US) DOWN ARROW  |

| Printable Characters |      |           |     |      |           |
|----------------------|------|-----------|-----|------|-----------|
| DEC                  | HEX  | CHARACTER | DEC | HEX  | CHARACTER |
| 32                   | 0x20 | <SPACE>   | 64  | 0x40 | @         |
| 33                   | 0x21 | !         | 65  | 0x41 | A         |
| 34                   | 0x22 | "         | 66  | 0x42 | B         |
| 35                   | 0x23 | #         | 67  | 0x43 | C         |
| 36                   | 0x24 | \$        | 68  | 0x44 | D         |
| 37                   | 0x25 | %         | 69  | 0x45 | E         |
| 38                   | 0x26 | &         | 70  | 0x46 | F         |
| 39                   | 0x27 | '         | 71  | 0x47 | G         |
| 40                   | 0x28 | (         | 72  | 0x48 | H         |
| 41                   | 0x29 | )         | 73  | 0x49 | I         |
| 42                   | 0x2A | *         | 74  | 0x4A | J         |
| 43                   | 0x2B | +         | 75  | 0x4B | K         |
| 44                   | 0x2C | ,         | 76  | 0x4C | L         |
| 45                   | 0x2D | -         | 77  | 0x4D | M         |
| 46                   | 0x2E | .         | 78  | 0x4E | N         |
| 47                   | 0x2F | /         | 79  | 0x4F | O         |
| 48                   | 0x30 | 0         | 80  | 0x50 | P         |
| 49                   | 0x31 | 1         | 81  | 0x51 | Q         |
| 50                   | 0x32 | 2         | 82  | 0x52 | R         |
| 51                   | 0x33 | 3         | 83  | 0x53 | S         |
| 52                   | 0x34 | 4         | 84  | 0x54 | T         |
| 53                   | 0x35 | 5         | 85  | 0x55 | U         |
| 54                   | 0x36 | 6         | 86  | 0x56 | V         |
| 55                   | 0x37 | 7         | 87  | 0x57 | W         |
| 56                   | 0x38 | 8         | 88  | 0x58 | X         |
| 57                   | 0x39 | 9         | 89  | 0x59 | Y         |
| 58                   | 0x3A | :         | 90  | 0x5A | Z         |
| 59                   | 0x3B | ;         | 91  | 0x5B | [         |
| 60                   | 0x3C | <         | 92  | 0x5C | \         |
| 61                   | 0x3D | =         | 93  | 0x5D | ]         |
| 62                   | 0x3E | >         | 94  | 0x5E | ^         |
| 63                   | 0x3F | ?         | 95  | 0x5F | _         |
|                      |      |           | 96  | 0x60 | <DEL>     |

# Extended ASCII

- ▶ **Extended ASCII** character encodings are eight-bit encodings that include the standard seven-bit ASCII characters, plus additional characters
- ▶ There are many extended ASCII encodings, that support different human languages

| Extended ASCII Characters |      |           |     |      |           |     |      |           |  |
|---------------------------|------|-----------|-----|------|-----------|-----|------|-----------|--|
| DEC                       | HEX  | CHARACTER | DEC | HEX  | CHARACTER | DEC | HEX  | CHARACTER |  |
| 128                       | 0x80 | €         | 171 | 0xAB | «         | 214 | 0xD6 | Ö         |  |
| 129                       | 0x81 | □         | 172 | 0xAC | ¬         | 215 | 0xD7 | ×         |  |
| 130                       | 0x82 | ,         | 173 | 0xAD |           | 216 | 0xD8 | Ø         |  |
| 131                       | 0x83 | ƒ         | 174 | 0xAE | ®         | 217 | 0xD9 | Ú         |  |
| 132                       | 0x84 | „         | 175 | 0xAF | °         | 218 | 0xDA | Ú         |  |
| 133                       | 0x85 | …         | 176 | 0xB0 | ◦         | 219 | 0xDB | Ú         |  |
| 134                       | 0x86 | †         | 177 | 0xB1 | ±         | 220 | 0xDC | Ú         |  |
| 135                       | 0x87 | ‡         | 178 | 0xB2 | ²         | 221 | 0xDD | Ý         |  |
| 136                       | 0x88 | ˜         | 179 | 0xB3 | ˜         | 222 | 0xDE | Þ         |  |
| 137                       | 0x89 | %o        | 180 | 0xB4 | ·         | 223 | 0xDF | Þ         |  |
| 138                       | 0x8A | Š         | 181 | 0xB5 | µ         | 224 | 0xE0 | à         |  |
| 139                       | 0x8B | ‹         | 182 | 0xB6 | ¶         | 225 | 0xE1 | á         |  |
| 140                       | 0x8C | Œ         | 183 | 0xB7 | ·         | 226 | 0xE2 | â         |  |
| 141                       | 0x8D | □         | 184 | 0xB8 | ,         | 227 | 0xE3 | ã         |  |
| 142                       | 0x8E | Ž         | 185 | 0xB9 | ı         | 228 | 0xE4 | ä         |  |
| 143                       | 0x8F | □         | 186 | 0xBA | º         | 229 | 0xE5 | å         |  |
| 144                       | 0x90 | □         | 187 | 0xBB | »         | 230 | 0xE6 | æ         |  |
| 145                       | 0x91 | ‘         | 188 | 0xBC | ¼         | 231 | 0xE7 | ç         |  |
| 146                       | 0x92 | ’         | 189 | 0xBD | ½         | 232 | 0xE8 | é         |  |
| 147                       | 0x93 | “         | 190 | 0xBE | ¾         | 233 | 0xE9 | é         |  |
| 148                       | 0x94 | ”         | 191 | 0xBF | ¸         | 234 | 0xEA | ê         |  |
| 149                       | 0x95 | •         | 192 | 0xC0 | À         | 235 | 0xEB | ë         |  |
| 150                       | 0x96 | —         | 193 | 0xC1 | Á         | 236 | 0xEC | í         |  |
| 151                       | 0x97 | —         | 194 | 0xC2 | Â         | 237 | 0xED | í         |  |
| 152                       | 0x98 | ˜         | 195 | 0xC3 | Ã         | 238 | 0xEE | î         |  |
| 153                       | 0x99 | ™         | 196 | 0xC4 | Ä         | 239 | 0xEF | î         |  |
| 154                       | 0x9A | Š         | 197 | 0xC5 | Å         | 240 | 0xF0 | ð         |  |
| 155                       | 0x9B | ›         | 198 | 0xC6 | Æ         | 241 | 0xF1 | ñ         |  |
| 156                       | 0x9C | œ         | 199 | 0xC7 | Ç         | 242 | 0xF2 | ò         |  |
| 157                       | 0x9D | □         | 200 | 0xC8 | É         | 243 | 0xF3 | ó         |  |
| 158                       | 0x9E | ž         | 201 | 0xC9 | É         | 244 | 0xF4 | ô         |  |
| 159                       | 0x9F | Ý         | 202 | 0xCA | Ê         | 245 | 0xF5 | ö         |  |
| 160                       | 0xA0 |           | 203 | 0xCB | Ê         | 246 | 0xF6 | ö         |  |
| 161                       | 0xA1 | í         | 204 | 0xCC | Í         | 247 | 0xF7 | +         |  |
| 162                       | 0xA2 | ¢         | 205 | 0xCD | Í         | 248 | 0xF8 | ø         |  |
| 163                       | 0xA3 | £         | 206 | 0xCE | Í         | 249 | 0xF9 | ú         |  |
| 164                       | 0xA4 | ¤         | 207 | 0xCF | Ï         | 250 | 0xFA | ú         |  |
| 165                       | 0xA5 | ¥         | 208 | 0xD0 | Ð         | 251 | 0xFB | û         |  |
| 166                       | 0xA6 | ₩         | 209 | 0xD1 | Ñ         | 252 | 0xFC | ü         |  |
| 167                       | 0xA7 | §         | 210 | 0xD2 | Ó         | 253 | 0xFD | ý         |  |
| 168                       | 0xA8 |           | 211 | 0xD3 | Ó         | 254 | 0xFE | þ         |  |
| 169                       | 0xA9 | ©         | 212 | 0xD4 | Ó         | 255 | 0xFF | ÿ         |  |
| 170                       | 0xAA | ª         | 213 | 0xD5 | Ó         |     |      |           |  |

# ASCII in Python

- ▶ The `\x` escape denotes a character encoded by the single-byte hex value given by the subsequent two characters
- ▶ For example, the capital letter ‘N’ has the value 78, which is 4E in hex, thus:

```
print('\x4e')
```

N

- ▶ The `ord()` function returns the ASCII value of a character and the `chr()` function returns the character represented by the ASCII value:

```
ord("a") # the ASCII value of character a
```

97

```
chr(65) # the character represented by ASCII value 65
```

'A'

## Exercise (12)

---

- ▶ Ask the user to enter a small English letter (a-z)
- ▶ Print its corresponding capital letter, e.g. K for k
- ▶ Ask the user to enter another character
- ▶ Print if the second character is a digit or not (i.e., one of the characters 0-9)

# Unicode

- ▶ Python 3 strings are composed of *Unicode* characters
- ▶ Unicode is a standard describing the representation of more than 140,000 characters in almost every human language as well as many other special characters
- ▶ Unicode assigns a number (**code point**) to every character
- ▶ These code points can be implemented as byte sequences by different character encodings (e.g., UTF-8, UTF-16, UTF-32)
- ▶ By default, Python 3 uses the **UTF-8** encoding, which is the most widely used today
- ▶ UTF-8 uses one byte for the first 128 code points, and up to 4 bytes for the others
  - ▶ The first 128 UTF-8 code points are the ASCII characters, which means that any ASCII text is also a UTF-8 text
- ▶ For a list of code points, see the official Unicode website's code charts at  
<http://www.unicode.org/charts/>

# Unicode Characters

- If your editor doesn't allow you to enter a Unicode character, you can use its 16- or 32-bit hex value or its Unicode character name:

```
'\u00E9' # 16-bit hex value
```

```
'é'
```

```
'\U000000E9' # 32-bit hex value
```

```
'é'
```

```
'\N{LATIN SMALL LETTER E WITH ACUTE}' # by name
```

```
'é'
```

- Python even supports Unicode variable names:

```
Σ = 4
```

```
Σ
```

```
4
```

- This is mostly a bad idea, because of the difficulty in entering non-ASCII characters from a standard keyboard

# Unicode Characters

- ▶ `ord(c)` returns an integer representing the Unicode code point of the character `c`

```
print(ord('a'))
print(ord('€'))
```

97  
8364

- ▶ `chr(i)` returns the string representing a character whose Unicode code point is the integer `i` (the inverse of `ord`)
  - ▶ The valid range for the argument is from 0 through 1,114,111 (0x10FFFF in base 16)

```
print(chr(97))
print(chr(8364))
```

a  
€

# String Length

- ▶ The `len()` built-in function counts the number of characters in the string
  - ▶ If the string is in Unicode, `len()` returns the number of Unicode characters

```
len("hello world")
```

11

```
s = "הוא אוסף אותו מהר בשלוש וחצי"
len(s)
```

28

- ▶ If you want to know the number of bytes needed to store the string in memory, you first have to convert it into a Python byte string using the `encode()` method

```
len(s.encode('utf-8'))
```

51

## Exercise (13)

- ▶ Ask the user to enter a card number between 1 and 14
- ▶ Print the symbol of the corresponding card from the Spades suite
- ▶ The Unicode characters of the symbol cards is given in the following table:

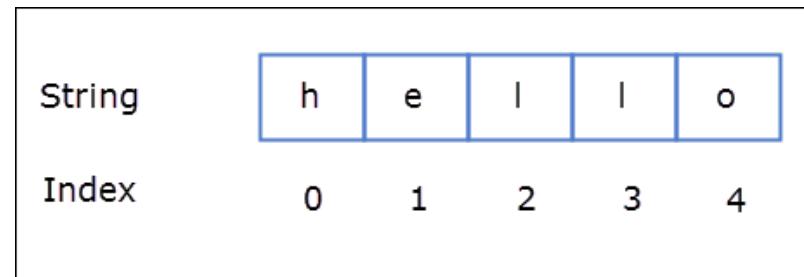
|         | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| U+1F0Ax | 🂠 | 🂡 | 🂢 | 🂣 | 🂤 | 🂥 | 🂦 | 🂧 | 🂨 | 🂩 | 🂪 | 🂫 | 🂬 | 🂭 | 🂮 |   |
| U+1F0Bx |   | 🂱 | 🂲 | 🂳 | 🂴 | 🂵 | 🂶 | 🂷 | 🂸 | 🂹 | 🂺 | 🂻 | 🂼 | 🂽 | 🂾 | 🂿 |
| U+1F0Cx |   | 🂱 | 🂲 | 🂳 | 🂴 | 🂵 | 🂶 | 🂷 | 🂸 | 🂹 | 🂺 | 🂻 | 🂼 | 🂽 | 🂾 | 🂿 |
| U+1F0Dx |   | 🂱 | 🂲 | 🂳 | 🂴 | 🂵 | 🂶 | 🂷 | 🂸 | 🂹 | 🂺 | 🂻 | 🂼 | 🂽 | 🂾 | 🂿 |
| U+1F0Ex | 🂱 | 🂲 | 🂳 | 🂴 | 🂵 | 🂶 | 🂷 | 🂸 | 🂹 | 🂺 | 🂻 | 🂼 | 🂽 | 🂾 | 🂿 |   |
| U+1F0Fx | 🂱 | 🂲 | 🂳 | 🂴 | 🂵 | 🂶 | 🂷 | 🂸 | 🂹 | 🂺 | 🂻 | 🂼 | 🂽 | 🂾 | 🂿 |   |

- ▶ For example: Enter a card number: 5



# Indexing Strings

- ▶ **Indexing** (or “subscripting”) a string returns a single character at a given location
- ▶ An index refers to the position of a character inside a string
- ▶ Like all sequences in Python, strings are zero-indexed
  - ▶ which means that the first character is at index 0, and the final character in a string consisting of  $n$  characters is at index  $n - 1$



- ▶ To access a character at index  $i$  in a string  $str$ , write the index number of the character inside square brackets [], like this:  $str[i]$ 
  - ▶ The character is returned in a str object of length 1

# Indexing Strings

## ► Examples:

```
s = "hello"
```

```
s[0] # get the first character
```

```
'h'
```

```
s[1] # get the second character
```

```
'e'
```

```
s[len(s) - 1] # get the last (fifth) character
```

```
'o'
```

- The last valid index for string s is 4
- Trying to access characters beyond the last valid index will raise an IndexError

# Negative Indexes

- ▶ We can also use negative indexes to access characters from the end of the string
- ▶ Negative index start from -1, so the index position of the last character is -1, and the index position of the first character is  $-n$  (for a string of length  $n$ )

```
s = "markdown"
s[-1] # get the last character
'n'
s[-2] # get the second last character
'w'
s[-len(s)] # get the first character
'm'
```

|                |    |    |    |    |    |    |    |    |
|----------------|----|----|----|----|----|----|----|----|
| Negative Index | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |
| String         | m  | a  | r  | k  | d  | o  | w  | n  |
| Index          | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  |

# Membership Operators

- ▶ **in** or **not in** operators are used to check if a string contains a given substring:

```
str1 = "object oriented"
```

```
"ted" in str1 # Does "ted" exist in str1?
```

True

```
"eject" in str1 # Does "eject" exist in str1?
```

False

```
"orion" not in str1 # Does "orion" doesn't exist in str1?
```

True

# Slicing Strings

- ▶ Slicing a string, `s[i:j]`, produces a substring of a string between the characters at two indexes, including the first (`i`) but excluding the second (`j`)
- ▶ If the first index is omitted, 0 is assumed
- ▶ If the second index is omitted, the string is sliced to its end

```
s = "markdown"
s[0:3] # get a slice from index 0 to 3, not including 3
```

```
'mar'
```

```
s[2:5] # get a slice from index 2 to 5, not including 5
```

```
'rkd'
```

```
s[:4] # start slicing from the beginning, same as s[0:4]
```

```
'mark'
```

```
s[5:] # slicing goes to the end of the string, same as s[5:len(s)]
```

```
'own'
```

```
s[:] # the same as s[0:len(s)]
```

```
'markdown'
```

# Slicing Strings

- ▶ Unlike indexing, slicing a string outside its bounds does not raise an error:

```
s[2:len(s)+10]
```

```
'rardown'
```

```
s[10:]
```

```
''
```

- ▶ We can also use negative index in string slicing:

```
s[1:-1] # slice the string from index 1 to index -1, not including -1
```

```
'arkdow'
```

# Slicing Strings

- ▶ The optional, third number in a slice specifies the *stride*
  - ▶ If omitted the default is 1: return every character in the requested range
  - ▶ To return every  $k^{\text{th}}$  letter, set the stride to  $k$
  - ▶ Negative values of  $k$  reverse the string

```
s = 'Markdown'
s[2:6:2]
```

'rd'

```
s[::-2]
```

'Mrdw'

```
s[-1:4:-1] # take the characters from the last (index-1)
down to (but not including) character at index 4
with stride -1 (select every character in reverse direction)
```

'nwo'

```
s[::-1] # reverse a string
```

'nwodkraM'

## Exercise (14)

▶ Slice the string `s='sehemewe'` to produce the following substrings:

- ▶ 'see'
- ▶ 'he'
- ▶ 'me'
- ▶ 'we'
- ▶ 'hem'
- ▶ 'meh'
- ▶ 'wee'

## Exercise (15)

- ▶ Get a string from the user and prints whether it is a palindrome
- ▶ A palindrome is a string that reads the same forward as backward
  - ▶ e.g., level, radar, racecar, madam, noon, civic
- ▶ Use a single-line expression for determining if the string is a palindrome

# String Methods

- ▶ String objects come with a large number of methods for manipulating and transforming
- ▶ These methods are accessed using the usual dot notation we've met already
- ▶ They can be grouped into the following categories:
  - ▶ Testing strings
  - ▶ Searching for a substring inside a string
  - ▶ Formatting strings
  - ▶ Converting strings

# Testing Strings

## ▶ Methods for testing strings:

| Method    | Description                                                                                                  |
|-----------|--------------------------------------------------------------------------------------------------------------|
| isalpha() | Returns True if all characters in the string are alphabetic; otherwise return False.                         |
| isdigit() | Returns True if all characters in the string are digits; otherwise return False.                             |
| isalnum() | Returns True if all characters in the string are alphanumeric (digits or alphabets); otherwise return False. |
| islower() | Returns True if all the characters in the string are in lowercase; otherwise return False.                   |
| isupper() | Returns True if all the characters in the string are in uppercase; otherwise return False.                   |
| isspace() | Returns True if all the characters in the string are whitespace characters ; otherwise return False.         |

# Testing Strings – Examples

```
"hello".isalpha()
```

True

```
"abc123".isalpha()
```

False

```
"2048".isdigit()
```

True

```
"101.29".isdigit()
```

False

```
"Abc123".isalnum()
```

True

```
"$##".isalnum()
```

False

```
"abc".islower()
```

True

```
s = "A bite of python"
s.islower()
```

False

```
s.isupper()
```

False

\n\t.isspace()

True

```
1 2 3.isspace()
```

False

# Searching in Strings

- ▶ Methods that allows you to search for a substring inside a string:

| Method                          | Description                                                                                                                                    |
|---------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>endswith(suffix)</code>   | Returns True if the string ends with the substring <i>suffix</i> ; otherwise return False.                                                     |
| <code>startswith(prefix)</code> | Returns True if the string starts with the substring <i>prefix</i> ; otherwise return False.                                                   |
| <code>find(substring)</code>    | Returns the lowest index in the string where <i>substring</i> is found. If <i>substring</i> is not found return -1.                            |
| <code>rfind(substring)</code>   | Returns the highest index in the string where <i>substring</i> is found. If <i>substring</i> is not found return -1.                           |
| <code>index(substring)</code>   | Returns the lowest index in the string where <i>substring</i> is found. If <i>substring</i> doesn't exist in the list, an exception is raised. |
| <code>count(substring)</code>   | Returns the number of occurrences of <i>substring</i> found in the string. If no occurrence is found return 0.                                 |

# Searching in Strings – Examples

```
s = "abc"
s.endswith("bc")
```

True

```
"python".startswith("py")
```

True

```
"Learning Python".find("n")
```

4

```
"Learning Python".find("at")
```

-1

```
"Learning Python".rfind("n")
```

14

```
"Learning Python".count("n")
```

3

# Manipulating Strings

- ▶ The following methods are used to return a modified version of the string:

| Method            | Description                                                                                                                                                            |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| lower()           | Returns a copy of the string with all characters in uppercase.                                                                                                         |
| upper()           | Returns a copy of the string with all characters in lowercase.                                                                                                         |
| capitalize()      | Returns a copy of the string after capitalizing only the first letter in the string.                                                                                   |
| title()           | Returns a copy of the string with all words starting with capitals and other characters in lowercase.                                                                  |
| lstrip([chars])   | Returns a copy of the string with leading characters specified by [chars] removed. If [chars] is omitted, any leading whitespace is removed.                           |
| rstrip([chars])   | Returns a copy of the string with trailing characters specified by [chars] removed. If [chars] is omitted, any trailing whitespace is removed.                         |
| strip([chars])    | Returns a copy of the string with leading and trailing characters specified by [chars] removed. If [chars] is omitted, any leading and trailing whitespace is removed. |
| replace(old, new) | Returns a copy of the string with each substring <i>old</i> replaced with <i>new</i> .                                                                                 |

# Manipulating Strings – Examples

```
"abcDEF".lower()
```

```
'abcdef'
```

```
"abc".lower()
```

```
'abc'
```

```
"ABCdef".upper()
```

```
'ABCDEF'
```

```
"a long string".capitalize()
```

```
'A long string'
```

```
"a long string".title()
```

```
'A Long String'
```

```
s1 = "\n\tName\tAge"
print(s1)
```

```
Name Age
```

```
s2 = s1.strip()
s2
```

```
'Name\tAge'
```

```
print(s2)
```

```
Name Age
```

```
s = "--Name\tAge--"
s.lstrip("-")
```

```
'Name\tAge--'
```

```
s1 = "Learning C"
s2 = s1.replace("C", "Python")
s2
```

```
'Learning Python'
```

# Formatting Strings

- ▶ The following table lists some formatting methods of strings:

| Method                 | Description                                                                                                |
|------------------------|------------------------------------------------------------------------------------------------------------|
| center( <i>width</i> ) | Returns a copy of the string after centering it in a string with total number of characters <i>width</i>   |
| ljust( <i>width</i> )  | Returns a copy of the string after left align it in a string with total number of characters <i>width</i>  |
| rjust( <i>width</i> )  | Returns a copy of the string after right align it in a string with total number of characters <i>width</i> |

```
"Name".center(10)
```

```
' Name '
```

```
"Name".ljust(10)
```

```
'Name '
```

```
"Name".rjust(10)
```

```
' Name '
```

## Exercise (16)

- Predict the results of the following statements and check them in Jupyter Notebook:

```
days = 'Sun Mon Tue Wed Thurs Fri Sat'
```

```
print(days[days.find('M'):])
print(days[days.find('M'):days.find('Sa')].rstrip())
print(days[6:3:-1].lower()*3)
print(days.replace('rs', '')[::-4])
```

# String Comparison

- ▶ We can compare strings using the relational operators
- ▶ Strings are compared using a **Lexicographical comparison**:
  - ▶ The strings are compared using the Unicode value of their corresponding characters
  - ▶ The comparison starts off by comparing the first character from both strings:
    - ▶ If they differ, the Unicode values of the corresponding characters are compared to determine the outcome of the comparison
    - ▶ If they are equal, the next two characters are compared
  - ▶ This process continues until either string is exhausted
  - ▶ If a short string appears at the start of another long string then the short string is smaller

```
"linker" > "linquish"
```

False

```
"ab" < "abc"
```

True

# The print() Function

- ▶ `print()` is a built-in function takes a list of objects, and also the optional arguments:
  - ▶ **end** – specify which characters should end the string
  - ▶ **sep** – specify which characters should be used to separate the printed objects
  - ▶ Omitting these additional arguments prints the object fields separated by a *single space* and the line is ended with a *newline* character

```
ans = 6
print("Solve:", 2, "x =", ans, "for x")
```

Solve: 2 x = 6 for x

```
print("Solve: ", 2, "x = ", ans, " for x", sep="", end="!\n")
```

Solve: 2x = 6 for x!

- ▶ To suppress the newline at the end of a printed string, specify `end=""`

```
print("A line with no newline character.", end="")
print("Another line")
```

A line with no newline character.Another line

# The print() Function

- ▶ `print()` can be used to create simple text tables:

```
title = "| Life Satisfaction Index |"
line = '+' + '-'*11 + '-'*15 + '+'
header = "| Country | Satisfaction |"

print(line,
 title,
 line,
 header,
 line,
 "| Australia | 7.3 |",
 "| Israel | 7.2 |",
 "| Spain | 6.4 |",
 line,
 sep="\n")
```

```
+-----+
| Life Satisfaction Index |
+-----+
| Country | Satisfaction |
+-----+
Australia	7.3
Israel	7.2
Spain	6.4
+-----+
```

# String Formatting

- ▶ It is possible to use a string's **format()** method to insert objects into it

```
"{} + {} = {}".format(2, 3, 5)
'2 + 3 = 5'
```

- ▶ The **format()** method is called on the string literal with the arguments 2, 3 and 5, which are interpolated, in order, into the locations of the **replacement fields**, indicated by braces {}
- ▶ Replacement fields can also be numbered (starting at 0) or named
  - ▶ Helps with longer strings and allows for the same value to be interpolated more than once

```
"{1} + {0} = {2}".format(2, 3, 5)
'3 + 2 = 5'
```

```
"{num1} + {num2} = {answer}".format(num1=2, num2=3, answer=5)
'2 + 3 = 5'
```

```
"{0} + {0} = {1}".format(2, 2 + 2)
'2 + 2 = 4'
```

# String Formatting

- ▶ Replacement fields can be given a minimum size within the string by the inclusion of an integer length after a colon as follows:

```
"==== {0:12} ====".format("Python")
```

```
'==== Python ==='
```

- ▶ If the string is too long for the minimum size, it will take up as many characters as needed
- ▶ The alignment of the interpolated string can be controlled with the single characters < (left), > (right) and ^ (center):

```
"==== {0:<12} ====".format("Python")
```

```
'==== Python ==='
```

```
"==== {0:>12} ====".format("Python")
```

```
'===== Python ==='
```

```
"==== {0:^12} ====".format("Python")
```

```
'==== Python ==='
```

## Exercise (17)

- ▶ What is the output of the following code? How does it work?

```
suff = "thstndrdththththththth"
n = 1
print("{}{}".format(n, suff[n*2:n*2+2]))
n = 3
print("{}{}".format(n, suff[n*2:n*2+2]))
n = 5
print("{}{}".format(n, suff[n*2:n*2+2]))
```

# Formatting Numbers

- ▶ The string `format()` method provides a powerful way to format numbers
- ▶ The specifiers ‘d’, ‘b’, ‘o’, ‘x’, ‘X’ indicate a decimal, binary, octal, lowercase hex, and uppercase hex integer, respectively:

```
x = 254
```

```
"x = {0:5}".format(x)
```

```
'x = 254'
```

```
"x = {0:10b}".format(x) # binary
```

```
'x = 11111110'
```

```
"x = {0:5o}".format(x) # octal
```

```
'x = 376'
```

```
"x = {0:5x}".format(x) # hex (Lowercase)
```

```
'x = fe'
```

```
"x = {0:5X}".format(x) # hex (uppercase)
```

```
'x = FE'
```

# Formatting Numbers

- ▶ By default, all types of numbers are right aligned
- ▶ We can change the default alignment by using following the characters < (left justify) and > (right justify)

```
"x = {0:<5}".format(x)
```

```
'x = 254 '
```

```
"x = {0:>5}".format(x)
```

```
'x = 254'
```

- ▶ Numbers can be padded with zeros to fill out the specified field size by prefixing the minimum width with a 0:

```
"x = {0:05}".format(x)
```

```
'x = 00254'
```

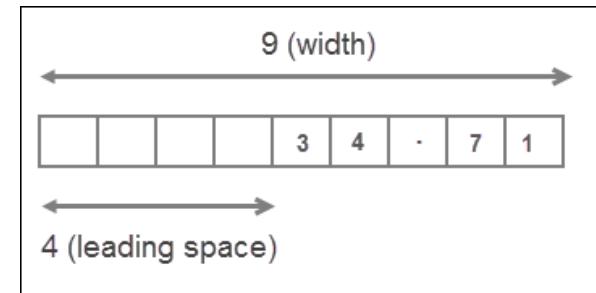
# Formatting Numbers

- To format floating point numbers use the following format specifier:

```
width.precision
```

- width** includes the digits before and after the decimal and the decimal point itself
  - precision** is the number of decimal places after the decimal point
  - The t character following the precision is the type code or specifier. Most useful specifiers:
    - 'f': fixed-point notation
    - 'e'/'E': exponent (i.e., "scientific" notation)
    - 'g'/'G': a general format which uses scientific notation for very large and very small numbers
- Example:

```
a = 34.71248
"{0:9.2f}".format(a)
' 34.71'
```



# Formatting Numbers

- ▶ We can also omit the width entirely, in which case it is automatically determined by the length of the value:

```
import math
"{0:.5f}".format(math.pi)

'3.14159'
```

- ▶ To format a number in Scientific Notation, just replace the type code from f to e or E:

```
"{0:10.3e}".format(5482.52291)

' 5.483e+03'
```

```
"{0:.2e}".format(0.0000212354)

'2.12e-05'
```

- ▶ You can separate thousands by commas by adding a comma , just after the width field or before the precision:

```
"{0:,.2f}".format(98813343817.71)

'98,813,343,817.71'
```

## Exercise (18)

- The table that follows gives the names, symbols, values, uncertainties and units of some physical constants

| Name                     | Symbol  | Value                         | Uncertainty           | Units                         |
|--------------------------|---------|-------------------------------|-----------------------|-------------------------------|
| Boltzmann constant       | $k_B$   | $1.3806504 \times 10^{-23}$   | $2.4 \times 10^{-29}$ | $\text{J K}^{-1}$             |
| Speed of light           | $c$     | 299792458                     | (def)                 | $\text{m s}^{-1}$             |
| Planck constant          | $h$     | $6.62606896 \times 10^{-34}$  | $3.3 \times 10^{-41}$ | $\text{J s}$                  |
| Avogadro constant        | $N_A$   | $6.02214179 \times 10^{23}$   | $3 \times 10^{16}$    | $\text{mol}^{-1}$             |
| Electron magnetic moment | $\mu_e$ | $-9.28476377 \times 10^{-24}$ | $2.3 \times 10^{-31}$ | $\text{J/T}$                  |
| Gravitational constant   | $G$     | $6.67428 \times 10^{-11}$     | $6.7 \times 10^{-15}$ | $\text{N m}^2 \text{kg}^{-2}$ |

- Defining variables of the form:

```

kB = 1.3806504e-23 # J/K
kB_unc = 2.4e-29 # uncertainty
kB_units = "J/K"

```

## Exercise (18) Cont.

- ▶ Use the string object's `format()` method to produce the following output:

```
kB = 1.381e-23 J/K

G = 0.000000000667428 Nm^2/kg^2

c = 2.9979e+08 m/s

==== G = +6.67E-11 [Nm^2/kg^2] ===

==== μe = -9.28E-24 [J/T] ===
```

- ▶ Hint: the Unicode codepoint for the lowercase Greek letter mu is U+03BC

## Exercise (19)

- ▶ Create the following table:

| Cereal Yields (kg/ha) |       |       |       |       |
|-----------------------|-------|-------|-------|-------|
| Country               | 1980  | 1990  | 2000  | 2010  |
| China                 | 2,937 | 4,321 | 4,752 | 5,527 |
| Germany               | 4,225 | 5,411 | 6,453 | 6,718 |
| United States         | 3,772 | 4,755 | 5,854 | 6,988 |

# Iterable Objects and Sequences

- ▶ **Iterable** is an object capable of returning its members one at a time
- ▶ Examples of iterables include all sequence types (such as list, str, and tuple) and some non-sequence types like dict, set, and file objects
- ▶ Objects of any classes you define with an `__iter__()` method, or with a `__getitem__()` method that implements Sequence semantics, are iterables
- ▶ Iterables can be used in a for loop and in many other places where a sequence is needed (e.g., `list()`, `map()`, `zip()`)
  
- ▶ A **sequence** is an iterable which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `__len__()` method that returns the length of the sequence
- ▶ Some built-in sequence types are list, str, tuple, range, and bytes

# Lists

- ▶ A Python list is an ordered, *mutable* and dynamically sized array of objects
  - ▶ In some other languages (like C and Java) this data structure is called an array
  - ▶ However, while arrays are of fixed size, lists grow automatically as needed
- ▶ A list is constructed by specifying its objects, separated by commas, between square brackets [], as follows:

```
variable = [item1, item2, item3, ..., itemN]
```

- ▶ For example:

```
numbers = [5, 88, 17, 35]
```

- ▶ Note that just like everything else, a list is an object too, of type list

```
type(numbers)
```

```
list
```

- ▶ The numbers variable only stores the address where the list object is actually stored in memory

# Lists

- ▶ To print the contents of a list just type the list name, or use the `print()` function:

```
numbers
```

```
[5, 88, 17, 35]
```

```
print(numbers)
```

```
[5, 88, 17, 35]
```

- ▶ A list can contain elements of different types:

```
mixed = ["a string", 3.14, [1, 2, 3], True]
mixed
```

```
['a string', 3.14, [1, 2, 3], True]
```

- ▶ To create an empty list simply type square brackets [] without any elements inside it
  - ▶ e.g., `list1 = []`

# Multi-Dimensional Lists

- ▶ The elements of a list can be lists themselves:

```
matrix = [
 [37, 88, 25], # first row
 [99, 31, 64] # second row
]
matrix
```

```
[[37, 88, 25], [99, 31, 64]]
```

- ▶ matrix contains two elements of type list

# Indexing

- ▶ The elements in a list are zero-indexed (like strings)
- ▶ That means that the first element is at index 0, second is at 1, third is at 2 and so on
- ▶ The last valid index will be one less than the length of the list
- ▶ We use the following syntax to access an element from the list: `my_list[index]`
- ▶ Examples:

```
list1 = [5, 24, 3.14, 16.5, 35]
list1[0] # get the first element
```

5

```
list1[1] # get the second element
```

24

```
list1[len(list1) - 1] # get the last element
```

35

# Indexing

- ▶ Trying to access an element beyond the last valid index will result in an IndexError:

```
list1[10]

IndexError: list index out of range
Traceback (most recent call last)
<ipython-input-18-1a2e5d318e75> in <module>()
----> 1 list1[10]

IndexError: list index out of range
```

- ▶ Just like strings, we can use negative indexes to access elements from the list end
  - ▶ Negative indexes start from -1

```
list1[-1] # get the last element
```

35

```
list1[-2] # get the second last element
```

16.5

```
list1[-len(list1)] # get the first element
```

5

# Membership Operators

- ▶ Just like strings, we can check whether an element exists in the list or not using the **in** and **not in** operators:

```
list1 = [3, "hello", True, 5.3]
5.3 in list1
```

True

```
"Hello" in list1
```

False

```
"joker" not in list1
```

True

# Lists and Mutability

- ▶ Lists are **mutable**, i.e., we can modify a list without creating a new list in the process
  - ▶ Unlike strings, which cannot be altered once defined
- ▶ For example, the items of a list can be reassigned:

```
list1 = ["str", "list", "int", "float"]
```

```
id(list1) # the address where the list is stored
```

```
2902477589384
```

```
list1[0] = "string" # update element at index 0
```

```
list1 # List1 has changed
```

```
['string', 'list', 'int', 'float']
```

```
id(list1) # the id remains the same
```

```
2902477589384
```

# Lists and Mutability

- Let's examine another example:

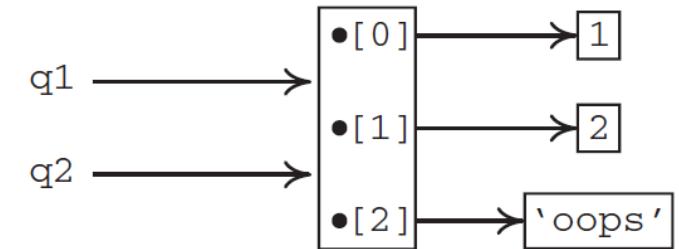
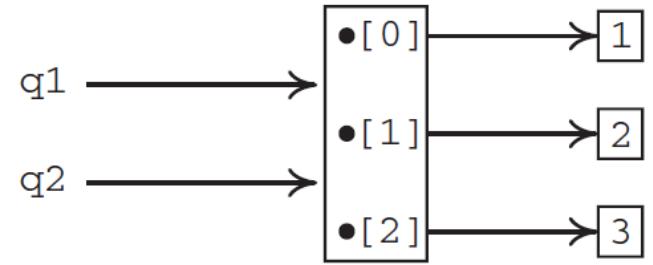
```
q1 = [1, 2, 3]
q2 = q1
```

```
q1[2] = "oops"
q1
```

```
[1, 2, 'oops']
```

```
q2
```

```
[1, 2, 'oops']
```



- The variables `q1` and `q2` refer to the *same list*, stored in the same memory location
- Because lists are mutable, the line `q1[2] = "oops"` actually changes one of the stored values at that location
- `q2` still points to the same location and so it appears to have changed as well

# Slicing Lists

- ▶ Lists can be sliced in the same way as string sequences:

```
list1 = [0, 0.1, 0.2, 0.3, 0.4, 0.5]
list1[1:4]
```

```
[0.1, 0.2, 0.3]
```

```
list1[::-1] # return a reversed copy of the list
```

```
[0.5, 0.4, 0.3, 0.2, 0.1, 0]
```

```
list1[1::2] # striding: returns elements at 1,3,5
```

```
[0.1, 0.3, 0.5]
```

- ▶ Taking a slice *copies the data* into a new list:

```
list2 = list1[1:4]
list2[1] = 999 # only affects list2
list2
```

```
[0.1, 999, 0.3]
```

```
list1
```

```
[0, 0.1, 0.2, 0.3, 0.4, 0.5]
```

# List Concatenation

- Like strings, lists can be joined using the + operator, which creates a new list containing the elements from both lists:

```
list1 = [1, 2, 3]
list2 = [11, 22, 33]
list3 = list1 + list2
list3
```

```
[1, 2, 3, 11, 22, 33]
```

- Another way to concatenate lists is to use the += operator, which modifies the existing list instead of creating a new one:

```
id(list1)
```

```
2902478184776
```

```
list1 += list2 # append list2 to list1
list1
```

```
[1, 2, 3, 11, 22, 33]
```

```
id(list1) # the address remains the same
```

```
2902478184776
```

# Repetition Operator

- ▶ We can use the \* operator with lists too
- ▶ The operator replicates the list:

```
actions = ["eat", "sleep", "repeat"]
daily_life = actions * 4
daily_life
```

```
['eat',
 'sleep',
 'repeat',
 'eat',
 'sleep',
 'repeat',
 'eat',
 'sleep',
 'repeat',
 'eat',
 'sleep',
 'repeat']
```

- ▶ This operator is useful for creating an empty list with a specific size, e.g.:

```
list1 = [None] * 10
list1
```

```
[None, None, None, None, None, None, None, None, None]
```

# List Built-in Functions

- The following table lists some functions, commonly used while working with lists:

| Function               | Description                                                |
|------------------------|------------------------------------------------------------|
| <code>len(list)</code> | Returns the number of elements in <i>list</i>              |
| <code>sum(list)</code> | Returns the sum of elements in the <i>list</i>             |
| <code>max(list)</code> | Returns the element with the greatest value in <i>list</i> |
| <code>min(list)</code> | Returns the element with the smallest value in <i>list</i> |

```
list1 = [1, 9, 4, 12, 82]
len(list1)
```

5

```
sum(list1)
```

108

```
max(list1)
```

82

```
min(list1)
```

1

# List Methods

- ▶ Just as for strings, Python lists come with a large number of useful methods:

| Method                              | Description                                                                                                                                                         |
|-------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>append(element)</code>        | Appends <i>element</i> to the end of the list                                                                                                                       |
| <code>extend(sequence)</code>       | Appends the elements of the <i>sequence</i> to the end of the list                                                                                                  |
| <code>index(element)</code>         | Returns the index of the first occurrence of <i>element</i> in the list. If <i>element</i> doesn't exist in the list, an exception is raised.                       |
| <code>insert(index, element)</code> | Inserts <i>element</i> at the specified <i>index</i>                                                                                                                |
| <code>pop([index])</code>           | Removes the element at the specified <i>index</i> and returns the element. If <i>index</i> is not specified, it removes and returns the last element from the list. |
| <code>reverse()</code>              | Reverses the list in place                                                                                                                                          |
| <code>remove(element)</code>        | Removes the first occurrence of <i>element</i> from the list                                                                                                        |
| <code>sort()</code>                 | Sorts the list in place (in ascending order)                                                                                                                        |
| <code>copy()</code>                 | Returns a copy of the list                                                                                                                                          |
| <code>count(element)</code>         | Returns the number of elements equal to the <i>element</i> in the list                                                                                              |
| <code>clear()</code>                | Removes all elements from the list                                                                                                                                  |

# List Methods - Examples

```
list1 = []
list1.append(4)
list1
```

```
[4]
```

```
list1.extend([6, 7, 8]) # append elements 6,7,8 to list1
list1
```

```
[4, 6, 7, 8]
```

```
list1.insert(1, 5) # insert 5 at index 1
list1
```

```
[4, 5, 6, 7, 8]
```

```
list1.remove(7) # remove item 7 from the list
list1
```

```
[4, 5, 6, 8]
```

```
list1.index(8) # the item 8 appears at index 3
```

```
3
```

# Sorting Lists

- ▶ The **sort()** method sorts the list *in place*, i.e., it changes the list object but doesn't return a value

```
list1 = [2, 0, 4, 3, 1]
list1.sort()
list1

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

- ▶ To get a sorted *copy of the list*, leaving it unchanged, use the **sorted()** built-in function:

```
list1 = ['a', 'e', 'A', 'c', 'b']
list2 = sorted(list1) # returns a new list
list2

['A', 'a', 'b', 'c', 'e']
```

```
list1 # the old list is unchanged

['a', 'e', 'A', 'c', 'b']
```

# Sorting Lists

- ▶ By default, `sort()` and `sorted()` order the items in an array in *ascending order*
- ▶ Set the optional argument `reverse=True` to sort the items in descending order:

```
a = [10, 5, 5, 2, 6, 1, 67]
sorted(a, reverse=True)

[67, 10, 6, 5, 5, 2, 1]
```

- ▶ Python 3, unlike Python 2, doesn't allow direct comparisons between different types, so it is an error to attempt to sort a list containing a mixture of types:

```
b = [5, '4', 2, 8]
b.sort()
```

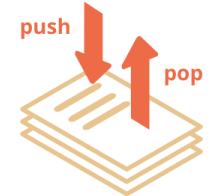
```

TypeError Traceback (most recent call last)
<ipython-input-82-d6e2bccdd6f3> in <module>()
 1 b = [5, '4', 2, 8]
----> 2 b.sort()

TypeError: '<' not supported between instances of 'str' and 'int'
```

# List as a Stack

- ▶ A **stack** is an ordered collection of elements which supports two operations:
  - ▶ push adds an element to the end
  - ▶ pop takes an element from the end
- ▶ The list methods **append()** and **pop()** make it very easy to use a list as a stack:
  - ▶ The end of the list is the top of the stack from which items may be added or removed



```
stack = []
stack.append(1)
stack.append(2)
stack.append(3)
print(stack)
```

```
[1, 2, 3]
```

```
stack.pop()
```

```
3
```

```
print(stack)
```

```
[1, 2]
```

# Split and join

- ▶ The string method **split()** generates a list of substrings from a given string, split on a specified separator:

```
months = "Jan Feb Mar Apr May Jun"
months.split() # By default, splits on whitespace
```

```
['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
```

```
fruits = "Apple,Banana,Melon,Orange"
fruits.split(",")
```

```
['Apple', 'Banana', 'Melon', 'Orange']
```

- ▶ The string method **join(sequence)** uses the string as a separator in joining a sequence of strings:

```
date = ["24", "6", "2018"]
"/".join(date)
```

```
'24/6/2018'
```

## Exercise (20)

- ▶ Read a sentence from the user and print it in reverse (i.e., reverse the words in the sentence)
- ▶ For example:

Please enter a sentence:

An apple a day keeps the doctor away

The sentence in reverse:

away doctor the keeps day a apple An

# Tuples

- ▶ A tuple is an *immutable* sequence of objects
  - ▶ i.e., you can't add, remove or modify its elements once it is created
  - ▶ As a result tuples are slightly faster for many uses than lists
- ▶ Tuples are constructed by placing the items inside parentheses:

```
t = (1, 2, 3, 4, 5)
t
```

```
(1, 2, 3, 4, 5)
```

```
t = ("alpha", "beta", "gamma")
t
```

```
('alpha', 'beta', 'gamma')
```

```
t0 = () # an empty tuple
t0
```

```
()
```

```
t1 = ("one",) # a singleton tuple
t1
```

```
('one',)
```

To create a tuple with just one item (a singleton), you must type a trailing comma after the item



# Operations on Tuples

- ▶ A tuple is essentially an immutable list
- ▶ As a result, most of the operations that can be performed on the list are also valid for tuples, such as:
  - ▶ Indexing or slicing using the [] operator
  - ▶ Built-in functions like len(), max(), min(), sum()
  - ▶ Membership operator in and not in
  - ▶ + and \* operators
  - ▶ Comparison operators
- ▶ However, tuples don't support operations that modify the tuple itself, such as:
  - ▶ Item assignment
  - ▶ Methods such as append(), insert(), remove(), reverse(), and sort()

# Operations on Tuples – Examples

```
t1 = ("alpha", "beta", "gamma")
len(t1) # Length of tuple
```

3

```
t1[1] # indexing
```

'beta'

```
t1[1:] # slicing
```

('beta', 'gamma')

```
"kappa" in t1 # membership
```

False

```
t1 * 2 # multiplication
```

('alpha', 'beta', 'gamma', 'alpha', 'beta', 'gamma')

```
t1 + ("delta",) # addition
```

('alpha', 'beta', 'gamma', 'delta')

# Packing and Unpacking

- ▶ Packing is a simple syntax which lets you create tuples "on the fly" without using parenthesis around the tuple's items:

```
t = 1, 2, 3
t
```

(1, 2, 3)

- ▶ You can also go the other way, and unpack the tuple into separate variables:

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

1 2 3

- ▶ Tuple unpacking is a common way of assigning multiple variables in one line:

```
a, b, c = 10, 20, 30
b
```

20

- ▶ The values 10, 20, 30 on the right-hand side are first packed into a tuple, which is then unpacked into the variables assigned on the left-hand side

# Swapping Variables

- ▶ Swapping values between two variables is a common programming operation
- ▶ In languages like C, to perform swapping you have to create an additional variable to store the data temporarily
- ▶ In Python, we can use tuples to swap the values of two variables:

```
Swapping values C-style
x, y = 10, 20
print(x, y)

tmp = x # now tmp contains 10
x = y # now x contains 20
y = tmp # now y contains 10
print(x, y)
```

```
10 20
20 10
```

```
Swapping values Python-style
x, y = 10, 20
print(x, y)

y, x = x, y
print(x, y)
```

```
10 20
20 10
```

# Iterable Objects

- ▶ Strings, lists and tuples are all examples for *iterable objects*
  - ▶ i.e., collections of objects which can be taken one at a time
- ▶ One way of seeing this is to use the alternative method of initializing a list (or tuple) using the built-in constructor methods **list()** and **tuple()**
  - ▶ These take any iterable object and generate a list or a tuple from its sequence of items
  - ▶ The data elements are copied in the construction of the new object

```
list("hello")
['h', 'e', 'l', 'l', 'o']
```

```
tuple([1, 2, 3, 4])
(1, 2, 3, 4)
```

- ▶ Because slices also return a copy of the elements in the sequence, the idiom `b = a[:]` is often used in preference to `b = list(a)`

# Iterable Unpacking Operator \*

- ▶ It is sometimes necessary to call a function with arguments taken from a list or other sequence
- ▶ The \* syntax, used in a function call, unpacks such a sequence into positional arguments to the function
- ▶ For example, the math.hypot function takes two arguments, a and b, and returns the quantity  $\sqrt{a^2 + b^2}$ .
- ▶ If the arguments you wish to use are in a list or tuple, the following will fail

```
t = [3, 4]
math.hypot(t)
```

```

TypeError Traceback (most recent call last)
<ipython-input-5-92482cb20c93> in <module>()
 1 t = [3, 4]
----> 2 math.hypot(t)

TypeError: hypot expected 2 arguments, got 1
```

# Iterable Unpacking Operator \*

- ▶ We could index the list explicitly to retrieve the two values we need:

```
t = [3, 4]
math.hypot(t[0], t[1])
```

5.0

- ▶ but a more elegant method is to unpack the list into arguments to the function:

```
math.hypot(*t)
```

5.0

- ▶ Python 3.5 extends the allowed uses of the \* operator in more cases, e.g., in expressions involving iterators/lists/tuples:

```
list1 = [1, 2, 3, *[4, 5, 6]]
list1
```

[1, 2, 3, 4, 5, 6]

## Exercise (21)

- Predict and explain the outcome of the following statements:

```
s = "hello"
a = [4, 10, 2]

print(s, sep="-")
print(*s, sep="-")
print(a)
print(*a, sep="\t")
list(*a,))
```

# Control Statements

- ▶ Control statements allow us to execute a set of statements only when certain conditions are met
- ▶ Python has the following control statements:
  - ▶ if statement
  - ▶ if-else statement
  - ▶ if-elif-else statement
  - ▶ while loop
  - ▶ for loop
  - ▶ break statement
  - ▶ continue statement

# if Statement

- ▶ The syntax of if statement is as follows:

```
if <boolean expression>:
 <indented statement 1>
 <indented statement 2>
 ...
<non-indented statements>
```

- ▶ If the boolean expression evaluates to true, then all the statements inside the if block are executed
  - ▶ Each statement in the if block must be indented by the same number of spaces
  - ▶ It is strongly recommended to use **four spaces** to indent code
- ▶ If the expression evaluates to false, then all the statements in the if block are skipped
  - ▶ And execution continues with the statements following the if block (which are not indented)

# if Statement

- ▶ The following program checks if the input number is greater than 10:

```
num = int(input("Enter a number: "))
if num > 10:
 print("The number is greater than 10")
```

```
Enter a number: 100
The number is greater than 10
```

- ▶ The **if** block can have any number of statements:

```
num = int(input("Enter a number: "))
if num > 10:
 print("Statement 1")
 print("Statement 2")
 print("Statement 3")
print("Executes every time you run the program")
print("Program ends here")
```

```
Enter a number: 45
Statement 1
Statement 2
Statement 3
Executes every time you run the program
Program ends here
```

# if-else Statement

- ▶ An if-else statement executes one set of statements when the condition is true and a different set of statements when the condition is false
- ▶ In this way, an if-else statement allows us to follow two courses of action
- ▶ The syntax of if-else statement is as follows:

```
if <boolean expression>:
 <statements 1>
else:
 <statements 2>
```

- ▶ When if-else statement executes, the boolean expression is tested:
  - ▶ if it evaluates to True then statements inside the if block are executed
  - ▶ if it evaluates to False then the statements in the else block are executed

# if-else Statement Examples

- ▶ A program to calculate the area and circumference of the circle:

```
radius = int(input("Enter radius: "))
if radius > 0:
 print("Circumference = ", 2 * 3.14 * radius)
 print("Area = ", 3.14 * radius ** 2)
else:
 print("Please enter a positive number")
```

```
Enter radius: 4
Circumference = 25.12
Area = 50.24
```

- ▶ A program to check the password entered by the user:

```
password = input("Enter a password: ")
if password == "secret":
 print("Welcome to the secret world")
else:
 print("Go home")
```

```
Enter a password: secret
Welcome to the secret world
```

# if-else Statement

- ▶ The test expressions doesn't have to evaluate explicitly to the boolean values True and False
- ▶ It is enough if they evaluate to a truthy or a falsy value
- ▶ For example:

```
num = int(input("Enter a number: "))
if num % 2:
 print(num, "is odd!")
else:
 print(num, "is even!")
```

```
Enter a number: 5
5 is odd!
```

- ▶ This works because  $\text{num} \% 2 = 1$  for odd integers, which is equivalent to True and  $\text{num} \% 2 = 0$  for even integers, which is equivalent to False

# Nested if Statements

- ▶ We can also write if or if-else statement inside another if or if-else statement
- ▶ For example, a program to find the largest of two numbers:

```
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))

if num1 > num2:
 print("The first number is greater")
else:
 if num1 < num2:
 print("The second number is greater")
 else:
 print("The numbers are equal")
```

```
Enter first number: 5
Enter second number: 7
The second number is greater
```

# if-elif-else Statement

- ▶ The if-elif-else statement is another variation of the if-else statement, which allows us to test multiple conditions easily instead of writing nested if-else statements
- ▶ The syntax of the if-elif-else statement is:

```
if <boolean expression 1>:
 <statements 1>
elif <boolean expression 2>:
 <statements 2>
elif <boolean expression 3>:
 <statements 3>
...
else:
 <statements>
```

- ▶ if <boolean expression 1> evaluates to True, <statements 1> are executed
- ▶ otherwise, if <boolean expression 2> evaluates to True, <statements 2> are executed, and so on
- ▶ if none of the preceding conditions evaluate to True, the statements in the block of code following else: are executed

# if-elif-else Example

- ▶ A program to determine a student grade based upon its score in the test:

```
score = int(input("Enter your test score: "))

if score >= 90:
 print("Excellent! Your grade is A")
elif score >= 80:
 print("Great! Your grade is B")
elif score >= 70:
 print("Good. Your grade is C")
elif score >= 60:
 print("Your grade is D. You should work harder.")
else:
 print("You failed in the exam")
```

Enter your test score: 87

Great! Your grade is B

# Conditional Expressions

- ▶ A conditional expression lets you write a single assignment statement that assigns a value to a variable which depends on the truthness of some condition

```
x = a if condition else b
```

- ▶ a is assigned to x if condition evaluates to true, and b is assigned to x otherwise
- ▶ This is equivalent to writing:

```
if condition:
 x = a
else:
 x = b
```

- ▶ Example:

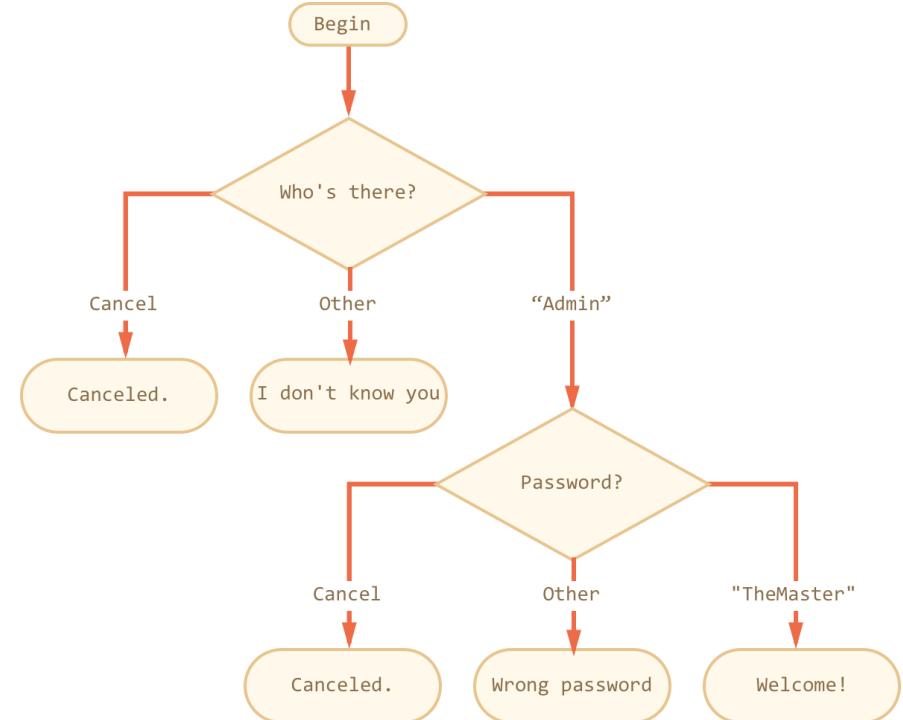
```
day = "Monday"
opening_time = 12 if day == "Sunday" else 9

opening_time
```

9

## Exercise (22)

- ▶ Write the code which asks for a login name
  - ▶ If the visitor enters “Admin”, then ask for a password
  - ▶ If the input is an empty line – print “Canceled”
  - ▶ If it’s other string – then print “I don’t know you”
- ▶ The password is checked as follows:
  - ▶ If it equals “TheMaster”, then show “Welcome!”
  - ▶ Other string – show “Wrong password”
  - ▶ For an empty string, show “Canceled”



## Exercise (23)

- ▶ In the Gregorian calendar a year is a *leap year* if it is divisible by 4, with the exception that years divisible by 100 are *not* leap years, unless they are also divisible by 400
  - ▶ For example, the years 1700 and 1800 were not leap years, but the years 1804 and 2000 were
- ▶ Write a program that lets the user check if a given year is a leap year or not

# Loops in Python

- ▶ A loop allows us to execute some set of statements multiple times
- ▶ Python provides two types loops:
  - ▶ while loop
  - ▶ for loop

# The while loop

- ▶ A while loop is a conditionally controlled loop, which executes a block of statements as long as the condition is true. Its syntax is:

```
while <boolean expression>:
 <indented statement 1>
 <indented statement 2>
 ...
 <non-indented statements>
```

- ▶ The indented group of statements is known as the **while block** or **loop body**
- ▶ The statements in the while block will keep executing as long as the condition is true
  - ▶ Each execution of the loop body is called **iteration**
- ▶ When the condition becomes false, the loop terminates and program control continues with the execution of the statement following the while block

# The while loop

- ▶ The following while loop calculates the sum of numbers between 1 and 10:

```
sum = 0

i = 1
while i < 11:
 sum += i
 i += 1

print("sum is", sum)
```

sum is 55

- ▶ This while loop executes until  $i < 11$
- ▶ The variable sum is used to accumulate the sum of numbers from 1 to 10
- ▶ In each iteration, the value of i is added to the variable sum and i is incremented by 1
- ▶ When i becomes 11, the loop terminates and the program control comes out of the while loop to execute the print() function at the last line

# The while loop

- ▶ The following program converts temperatures from Fahrenheit to Celsius, until the user decides to exit by answering ‘n’:

```
A program that converts tempratures from Fahrenheit to Celsius
keep_calculating = True

while keep_calculating:
 fah = int(input("Enter temprature in Fahrenheit: "))
 cel = (fah - 32) * 5/9
 print(format(fah, "0.2f") + "°F is same as", format(cel, "0.2f") + "°C\n")

 keep_calculating = input("Want to calculate more? Press y for Yes, n for No: ") == 'y'
```

```
Enter temprature in Fahrenheit: 60
60.00°F is same as 15.56°C
```

```
Want to calculate more? Press y for Yes, n for No: y
Enter temprature in Fahrenheit: 75
75.00°F is same as 23.89°C
```

```
Want to calculate more? Press y for Yes, n for No: n
```

## Exercise (24)

- ▶ Write a program that asks the user to enter a number and prints the sum of its digits
- ▶ For example, if the input is the number 8402 your program should print 14

## Exercise (25)

- ▶ The *Fibonacci sequence* is the sequence of numbers generated by applying the rules:  
$$a_1 = a_2 = 1, a_i = a_{i-1} + a_{i-2}$$
- ▶ That is, the  $i^{\text{th}}$  Fibonacci number is the sum of the previous two: 1, 1, 2, 3, 5, 8, 13, ...
- ▶ Write a program that gets a number  $n$  from the user, and prints all the Fibonacci numbers which are less than or equal to  $n$
- ▶ For example:

```
Enter the maximum for Fibonacci numbers: 500
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377
```

# The for loop

- ▶ It is often necessary to take the items in an iterable object one by one and do something with each in turn
- ▶ Other languages, such as C, use for loops to refer to each item in turn by its integer index
- ▶ In Python the more natural and convenient way is with the idiom:

```
for item in iterable object:
 # loop body
 <indented statements>
```

- ▶ The loop executes a block of statements (the loop body) for each item of the iterable object

# The for loop – Examples

```
iterating over a list
fruit_list = ['apple', 'melon', 'banana', 'orange']
for fruit in fruit_list:
 print(fruit, end=" ")
```

apple melon banana orange

```
iterating over a tuple
for i in (35, 22, 78, 5, 123):
 print(i, end=" ")
```

35 22 78 5 123

```
iterating over a string
for c in "hello":
 print(c, end=",")
```

h,e,l,l,o,

# Nested Loops

- ▶ Loops can be nested – the inner loop block needs to be indented by the same amount of whitespace as the outer loop (i.e. eight spaces):

```
fruit_list = ['apple', 'melon', 'banana', 'orange']
for fruit in fruit_list:
 for letter in fruit:
 print(letter, end=".")
```

```
a.p.p.l.e.
m.e.l.o.n.
b.a.n.a.n.a.
o.r.a.n.g.e.
```

- ▶ In this example, we iterate over the string items in `fruit_list` one by one, and for each string (fruit name), iterate over its letters

# The range Type

- ▶ The **range** type represents a sequence of numbers, which is generally used to iterate over with for loops
  - ▶ It would have been possible to create a list to hold the numbers, but this is memory inefficient
- ▶ A range object can be constructed with up to three arguments defining the first integer, the integer to stop at and the stride (the space between values)

```
range([a0=0], m, [stride=1])
```

- ▶ If the initial value  $a_0$  is not given it is taken to be 0
- ▶ stride is also optional and if it is not given it is taken to be 1
- ▶ stride can be negative
- ▶ The range represents the arithmetic progression  $a_n = a_0 + nd$  for  $n = 0, 1, 2, \dots$
- ▶  $d$  is the stride

# The range Type

- ▶ Examples:

```
a = range(5) # 0,1,2,3,4
b = range(1, 6) # 1,2,3,4,5
c = range(0, 6, 2) # 0,2,4
d = range(10, 0, -2) # 10,8,6,4,2
```

- ▶ In Python 3, the object created by range is not a list, but rather it is an iterable object that can produce integers on demand
- ▶ range objects can be indexed, cast into lists and tuples, and iterated over:

```
d[0]
```

```
10
```

```
d[1]
```

```
8
```

# Creating Lists using range() Function

- ▶ The range() function can be used to create long lists, by passing the range object to the list() constructor function
  - ▶ The list() function uses the numbers from the range sequence to create a list

```
list1 = list(range(5))
list1
```

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

```
list2 = list(range(0, 100, 5))
list2
```

```
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70,
75, 80, 85, 90, 95]
```

# For loop with range()

## ► Examples:

```
for i in range(5):
 print(i, end=" ")
```

0 1 2 3 4

```
for i in range(90, 99):
 print(i, end=" ")
```

90 91 92 93 94 95 96 97 98

```
for i in range(10, 20, 3):
 print(i, end=" ")
```

10 13 16 19

```
for i in range (20, 10, -2):
 print(i, end=" ")
```

20 18 16 14 12

# For loop with range()

- The following program uses a for loop to generate the squares of numbers from 1 to 20:

```
print("Number\t| Square")
print("-----")

for num in range(1, 21):
 print(num, "\t|", num * num)
```

| Number |  | Square |
|--------|--|--------|
| 1      |  | 1      |
| 2      |  | 4      |
| 3      |  | 9      |
| 4      |  | 16     |
| 5      |  | 25     |
| 6      |  | 36     |
| 7      |  | 49     |
| 8      |  | 64     |
| 9      |  | 81     |
| 10     |  | 100    |
| 11     |  | 121    |
| 12     |  | 144    |
| 13     |  | 169    |
| 14     |  | 196    |
| 15     |  | 225    |
| 16     |  | 256    |
| 17     |  | 289    |
| 18     |  | 324    |
| 19     |  | 361    |
| 20     |  | 400    |

# Break Statement

- ▶ The **break** statement is used to terminate the loop prematurely when a certain condition is met
- ▶ When a break statement is encountered inside the body of the loop, the current iteration stops and program control jumps to the statements following the loop:

```
for i in range(1, 10):
 if i == 5: # when i 5, exit the Loop
 break
 print("i =", i)
print("End of program")
```

```
i = 1
i = 2
i = 3
i = 4
End of program
```

# Break Statement

- ▶ Similarly, to find the index of the first occurrence of a negative number in a list:

```
list1 = [0, 4, 5, -2, 5, 10]
for idx, num in enumerate(list1):
 if num < 0:
 break
print(num, "occurs at index", idx)
```

-2 occurs at index 3

- ▶ Note that after escaping from the loop, the variables idx and num have the values that they had within the loop at the break statement

# Break Statement

- In a nested loop the break statement only terminates the loop in which it appears

```
for i in range(1, 5):
 print("Outer loop: i =", i)
 for j in range (10, 15):
 print("\tInner loop: j =", j)
 if j == 12:
 print("\t\tBreaking out of inner loop")
 break
```

```
Outer loop: i = 1
 Inner loop: j = 10
 Inner loop: j = 11
 Inner loop: j = 12
 Breaking out of inner loop
Outer loop: i = 2
 Inner loop: j = 10
 Inner loop: j = 11
 Inner loop: j = 12
 Breaking out of inner loop
Outer loop: i = 3
 Inner loop: j = 10
 Inner loop: j = 11
 Inner loop: j = 12
 Breaking out of inner loop
Outer loop: i = 4
 Inner loop: j = 10
 Inner loop: j = 11
 Inner loop: j = 12
 Breaking out of inner loop
```

# Continue Statement

- ▶ The **continue** statement is used to move ahead to the next iteration without executing the remaining statement in the body of the loop

```
for i in range(1, 10):
 if i % 2 == 1:
 continue
 print("i =", i)
```

```
i = 2
i = 4
i = 6
i = 8
```

- ▶ if  $i$  is not divisible by 2 (and hence  $i \% 2 == 1$ ), that loop iteration is canceled and the loop resumed with the next value of  $i$  (the print statement is skipped)

# Continue Statement

- ▶ We can also use break and continue statement together in the same loop:

```
while True:
 value = input("Enter a number: ")
 if value == 'q': # if input is 'q' exit from the Loop
 break
 if not value.isdigit(): # if input is not a digit move to the next iteration
 print("Enter digits only\n")
 continue

 value = int(value)
 print("Cube of", value, "is", value**3, "\n")
```

Enter a number: 5

Cube of 5 is 125

Enter a number: @

Enter digits only

Enter a number: 11

Cube of 11 is 1331

Enter a number: q

# Pass Statement

- ▶ The pass command does nothing (a “null” statement)
- ▶ It is useful as a “stub” for code that has not yet been written but where a statement is syntactically required by Python’s whitespace convention

```
for i in range(1, 11):
 if i == 6:
 pass # do something special if i is 6
 if i % 3 == 0:
 print(i, "is divisible by 3")
```

```
3 is divisible by 3
6 is divisible by 3
9 is divisible by 3
```

## else block

- ▶ A for or while loop may be followed by an **else** block of statements, which will be executed only if the loop finished “normally” (without the intervention of a break)
- ▶ For for loops, this means these statements will be executed after the loop has reached the end of the sequence it is iterating over
- ▶ For while loops, they are executed when the while condition becomes False
- ▶ For example, consider again our program to find the first occurrence of a negative number in a list. It behaves oddly if there aren’t any negative numbers in the list:

```
list1 = [0, 4, 5, 2, 5, 10]
for idx, num in enumerate(list1):
 if num < 0:
 break
print(num, "occurs at index", idx)
```

10 occurs at index 5

- ▶ It outputs the index and number of the last item in the list (whether it is negative or not)

# else block

- ▶ A way to improve this is to notice when the for loop runs through every item without encountering a negative number (and hence the break) and output a message:

```
list1 = [0, 4, 5, 2, 5, 10]
for idx, num in enumerate(list1):
 if num < 0:
 print(num, "occurs at index", idx)
 break
else:
 print("no negative numbers in the list")
```

no negative numbers in the list

- ▶ As another example, the following routine checks if a given number is prime or not:

```
import math
num = int(input("Enter a number: "))

for i in range(2, int(math.sqrt(num)) + 1):
 if num % i == 0:
 print(num, "is not prime")
 break
else:
 print(num, "is prime!")
```

Enter a number: 97  
97 is prime!

## Exercise (26)

---

- ▶ Get from the user two numbers: low and high
- ▶ Output all the even numbers between low and high (note that low and high themselves might be odd numbers)
- ▶ For example, if the user enters low = 5 and high = 14, you should print the numbers 6,8,10,12,14

## Exercise (27)

- ▶ Use a for loop to calculate  $\pi$  from the first 20 terms of the *Madhava series*:

$$\pi = \sqrt{12} \left( 1 - \frac{1}{3 \cdot 3} + \frac{1}{5 \cdot 3^2} - \frac{1}{7 \cdot 3^3} + \dots \right)$$

## Exercise (28)

- ▶ Get from the user a number
- ▶ Print to the console a square of stars whose length is the number specified by the user
- ▶ For example, if the user entered the number 8, your should print:

```



```

## Exercise (29)

- ▶ The *Luhn algorithm* is a simple checksum formula used to validate credit card and bank account numbers
- ▶ The algorithm may be written as the following steps:
  - ▶ Reverse the number
  - ▶ Treating the number as an array of digits, take the even-indexed digits (where the indexes *start at 1*) and double their values
  - ▶ If a doubled digit results in a number greater than 10, add the two digits (e.g., the digit 6 becomes 12 and hence  $1 + 2 = 3$ )
  - ▶ Sum this modified array
  - ▶ If the sum of the array modulo 10 is 0 the credit card number is valid
- ▶ Write a Python program to take a credit card number as a string of digits (possibly in groups, separated by spaces) and establish if it is valid or not
- ▶ For example, the string '4799 2739 8713 6272' is a valid credit card number, but any number with a single digit in this string changed is not

# enumerate()

- ▶ We cannot modify items in the list while iterating over it like this:

```
grades = [75, 83, 92, 86, 97]

for g in grades:
 g *= 1.05 # g holds copies of the values in the list
grades
```

```
[75, 83, 92, 86, 97]
```

- ▶ Changing the value of g in the loop body doesn't affect the elements in the list
- ▶ It is tempting to use the range() function to provide the indexes of the list like this:

```
for i in range(len(grades)):
 grades[i] *= 1.05
print(grades)
```

```
[78.75, 87.15, 96.6, 90.3, 99.75]
```

# enumerate()

- ▶ This works, of course, but it is more natural to avoid the explicit construction of a range object (and the call to the len built-in) by using **enumerate()**
- ▶ This method takes an iterable object and produces, for each item in turn, a tuple (index, item), consisting of a counting index and the item itself:

```
grades = [75, 83, 92, 86, 95]
for i, grade in enumerate(grades):
 grades[i] = grade * 1.05
print(grades)
```

```
[78.75, 87.15, 96.6, 90.3, 99.75]
```

- ▶ Each (index, item) tuple is unpacked in the for loop into the variables i and grade
- ▶ It is also possible to set the starting value of index to something other than 0:

```
mammals = ['kangaroo', 'wombat', 'platypus']
list(enumerate(mammals, 4))
```

```
[(4, 'kangaroo'), (5, 'wombat'), (6, 'platypus')]
```

## zip()

- ▶ What if you want to iterate over two (or more) sequences at the same time?
- ▶ The **zip()** built-in function creates an iterator object, in which each item is a tuple of items taken in turn from the sequences passed to it:

```
a = [1, 2, 3, 4]
b = ['a', 'b', 'c', 'd']
zip(a, b)
```

```
<zip at 0x23bc19420c8>
```

```
for pair in zip(a, b):
 print(pair)
```

```
(1, 'a')
(2, 'b')
(3, 'c')
(4, 'd')
```

```
list(zip(a, b)) # convert to List
```

```
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

## zip()

- ▶ A nice feature of zip is that it can be used to *unzip* sequences of tuples as well:

```
z = zip(a, b) # z generates (1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')
A, B = zip(*z) # zip((1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'))
print(A, B)
```

```
(1, 2, 3, 4) ('a', 'b', 'c', 'd')
```

```
list(A) == a, list(B) == b
```

```
(True, True)
```

- ▶ zip does not copy the items into a new object, so it is memory-efficient and fast
  - ▶ but this means that you only get to iterate over the zipped items once and you can't index it

```
z = zip(a, b)
print(list(z))

for pair in z:
 print(pair) # nothing: we've already exhausted the iterator z
```

```
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

## Exercise (31)

- ▶ Sorting a list of tuples arranges them in order of the first element in each tuple first
- ▶ If two or more tuples have the same first element, they are ordered by the second element, and so on:

```
sorted([(3, 1), (1, 4), (3, 0), (2, 2), (1, -1)])
```

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

- ▶ This suggests a way of using zip to sort one list using the elements of another
- ▶ Implement this method on the data below to produce an ordered list of the average amount of sunshine in hours in London by month
- ▶ Output the sunniest month first

| Jan   | Feb   | Mar   | Apr   | May   | Jun   |
|-------|-------|-------|-------|-------|-------|
| 44.7  | 65.4  | 101.7 | 148.3 | 170.9 | 171.4 |
| Jul   | Aug   | Sep   | Oct   | Nov   | Dec   |
| 176.7 | 186.1 | 133.9 | 105.4 | 59.6  | 45.8  |

# List Comprehension

- ▶ A list comprehension is a construct for creating a list based on another iterable object in a single line of code
- ▶ The syntax of list comprehension is:  

```
[expression for item in iterable]
```

  - ▶ In each iteration of the for loop an item is assigned a value from the iterable object, and the result of the expression is then used to produce values for the list
- ▶ For example, given a list of numbers, list1, a list of the squares of those numbers may be generated as follows:

```
list1 = [1, 2, 3, 4, 5, 6]
```

```
list2 = [x**2 for x in list1]
list2
```

```
[1, 4, 9, 16, 25, 36]
```

```
the same as:
list2 = []
for x in list1:
 list2.append(x**2)
list2
```

```
[1, 4, 9, 16, 25, 36]
```

- ▶ This is a faster and syntactically nicer way of creating the same list with the for loop on the right

# List Comprehension

- ▶ List comprehensions can also contain conditional statements, as follows:

```
[expression for item in iterable if condition]
```

- ▶ The only difference is that the expression before the for keyword is evaluated only when the condition is True
- ▶ For example, the following list comprehension generates a list of the squares of the even numbers in list1:

```
list2 = [x**2 for x in list1 if x % 2 == 0]
list2
```

```
[4, 16, 36]
```

- ▶ You can also use an if .. else expression, which must appear before the for loop:

```
[x**2 if x % 2 == 0 else x**3 for x in list1]
```

```
[1, 4, 27, 16, 125, 36]
```

- ▶ This comprehension squares the even integers and cubes the odd integers in list1

# List Comprehension

- ▶ The sequence used to construct the list can be any iterable object:

```
[x**3 for x in range(1, 10)]
```

```
[1, 8, 27, 64, 125, 216, 343, 512, 729]
```

```
[w.upper() for w in "hello"]
```

```
['H', 'E', 'L', 'L', 'O']
```

- ▶ Finally, list comprehensions can be nested
- ▶ For example, the following code flattens a list of lists:

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

```
[e for row in vlist for e in row]
```

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

## Exercise (32)

- ▶ Consider the lists:

```
a = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
b = [4, 2, 6, 1, 5, 0, 3]
```

- ▶ Predict and explain the output of the following statements:

```
[a[x] for x in b]
[a[x] for x in sorted(b)]
[a[b[x]] for x in b]
[x for (y,x) in sorted(zip(b,a))]
```

## Exercise (33)

- ▶ What does the following code do and how does it work?

```
nmax = 5
x = [1]
for n in range(1, nmax + 2):
 print(x)
 x = [[0] + x][i] + (x + [0])[i] for i in range(n + 1)]
```

## Exercise (34)

- ▶ Write a function **trace(*M*)** that uses list comprehension to calculate the *trace* of the matrix *M* (that is, the sum of its diagonal elements)
- ▶ Hint: the **sum()** built-in function takes an iterable object and sums its values
- ▶ Example:

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

```
trace(M)
```

15

## Exercise (35)

- ▶ The ROT13 substitution cipher encodes a string by replacing each letter with the letter 13 letters after it in the alphabet (cycling around if necessary)
  - ▶ For example, a→n and p→c
- ▶ Write a function **rot13\_word(word)** that gets a word expressed as a string of lowercase characters, and uses list comprehension to construct the ROT13-encoded version of that string
  - ▶ Hint: use the Python functions `ord()` and `chr()`
- ▶ Write a function **rot13\_sentence(sentence)** that encodes sentences of words (in lowercase) separated by spaces into a ROT13 sentence (in which the encoded words are also separated by spaces)
  - ▶ Use list comprehension and the previous function for this task

```
rot13_sentence("hello world")
```

```
'uryyb jbeyq'
```

# Functions

- ▶ A function is a named set of statements, which perform a specific task, and can be run more than once in a program
  - ▶ So far, we've been using many of Python's built-in functions such as `input()`, `print()`, `list()`, etc.
- ▶ There are two main advantages to using functions:
  - ▶ They enable code to be reused without having to be replicated in different parts of the program
  - ▶ They enable complex tasks to be broken into separate procedures, each implemented by its own function, thus making the code much easier to maintain and understand

# Defining and Calling Functions

- ▶ The syntax for defining a function is:

```
def function_name(param1, param2, ...):
 <indented statement 1>
 <indented statement 2>
 ...
 <indented statement n>
 <return statement>
```

- ▶ A function consists of two parts: a header and a body
- ▶ The function header starts with the **def** keyword, followed by name of the function, its arguments and ends with a colon (:)
- ▶ The function body contains statements which define what the function does
  - ▶ All the statements in the body of the function must be equally indented
  - ▶ If at any point during the execution of this statement block a **return statement** is encountered, the specified values are returned to the caller

# Defining and Calling Functions

- ▶ For example, the following function gets a number and returns it squared:

```
def square(x):
 x_squared = x ** 2
 return x_squared
```

- ▶ Once defined, the function can be called any number of times
- ▶ The syntax of calling a function is: `function_name(arg1, arg2, arg3, ..., argN)`
- ▶ The arguments passed at the function invocation (**actual parameters**), are bound to the names of the parameters defined in the function header (**formal parameters**)

```
num = 2
num_squared = square(num)
print(num, "squared is", num_squared)
```

2 squared is 4

```
print("8 squared is", square(8))
```

8 squared is 64

# Defining and Calling Functions

- ▶ Function definitions can appear anywhere in a Python program, but a function cannot be referenced or called before it is defined:

```
print("8 squared is", square(8))
def square(x):
 x_squared = x ** 2
 return x_squared
```

```
NameError Traceback (most recent call last)
<ipython-input-1-8e84aa8742da> in <module>()
----> 1 print("8 squared is", square(8))
 2 def square(x):
 3 x_squared = x ** 2
 4 return x_squared

NameError: name 'square' is not defined
```

# Returning a Value

- ▶ It is not necessary for a function to explicitly return any object
- ▶ If the function body doesn't have any return statement, or if the function falls off the end of its intended block without encountering a return statement, then Python's special value `None` is returned

```
def add(num1, num2):
 print("Sum is", num1 + num2)
```

```
result = add(10, 20)
print(result)
```

Sum is 30  
None

# Returning a Value

- ▶ To return two or more values from a function, pack them into a tuple
- ▶ For example, the following program defines a function to return both roots of the quadratic equation  $ax^2 + bx + c$  (assuming it has two real roots):

```
import math

def solve_quadratic(a, b, c):
 d = b**2 - 4*a*c;
 r1 = (-b + math.sqrt(d)) / (2*a)
 r2 = (-b - math.sqrt(d)) / (2*a)
 return r1, r2
```

```
solve_quadratic(1, -1, -6)
```

```
(3.0, -2.0)
```

# Functions as Objects

- ▶ In Python, functions are “first class” objects: they can have variables assigned to them, they can be passed as arguments and returned from other functions
- ▶ A function is given a name when it is defined, but that name can be reassigned to refer to a different object if desired (don’t do this unless you mean to!)
  - ▶ That’s the reason why you should not define variables such as range, list, str, etc.
- ▶ The following example associates an existing function with a new variable identifier:

```
import math
cosine = math.cos
id(math.cos)
```

2683242898056

```
id(cosine)
```

2683242898056

```
cosine(0)
```

1.0

# Naming Functions

---

- ▶ A function name should clearly describe what the function does
- ▶ When we see a function call in the code, a good name instantly gives us an understanding what it does and returns
- ▶ A function is an action, thus it is a widespread practice to start a function with a verbal prefix which vaguely describes the action
- ▶ For instance, functions starting with...
  - ▶ "show..." – usually show something
  - ▶ "get..." – return a value
  - ▶ "calc..." – calculate something
  - ▶ "create..." – create something
  - ▶ "check..." – check something and return a boolean

# Built-in Functions

- ▶ The Python interpreter has a number of functions and types built into it that are always available
- ▶ Be careful not to name your functions with the same name as a built-in function

| Built-in Functions         |                          |                           |                           |                             |
|----------------------------|--------------------------|---------------------------|---------------------------|-----------------------------|
| <code>abs()</code>         | <code>delattr()</code>   | <code>hash()</code>       | <code>memoryview()</code> | <code>set()</code>          |
| <code>all()</code>         | <code>dict()</code>      | <code>help()</code>       | <code>min()</code>        | <code>setattr()</code>      |
| <code>any()</code>         | <code>dir()</code>       | <code>hex()</code>        | <code>next()</code>       | <code>slice()</code>        |
| <code>ascii()</code>       | <code>divmod()</code>    | <code>id()</code>         | <code>object()</code>     | <code>sorted()</code>       |
| <code>bin()</code>         | <code>enumerate()</code> | <code>input()</code>      | <code>oct()</code>        | <code>staticmethod()</code> |
| <code>bool()</code>        | <code>eval()</code>      | <code>int()</code>        | <code>open()</code>       | <code>str()</code>          |
| <code>breakpoint()</code>  | <code>exec()</code>      | <code>isinstance()</code> | <code>ord()</code>        | <code>sum()</code>          |
| <code>bytearray()</code>   | <code>filter()</code>    | <code>issubclass()</code> | <code>pow()</code>        | <code>super()</code>        |
| <code>bytes()</code>       | <code>float()</code>     | <code>iter()</code>       | <code>print()</code>      | <code>tuple()</code>        |
| <code>callable()</code>    | <code>format()</code>    | <code>len()</code>        | <code>property()</code>   | <code>type()</code>         |
| <code>chr()</code>         | <code>frozenset()</code> | <code>list()</code>       | <code>range()</code>      | <code>vars()</code>         |
| <code>classmethod()</code> | <code>getattr()</code>   | <code>locals()</code>     | <code>repr()</code>       | <code>zip()</code>          |
| <code>compile()</code>     | <code>globals()</code>   | <code>map()</code>        | <code>reversed()</code>   | <code>__import__()</code>   |
| <code>complex()</code>     | <code>hasattr()</code>   | <code>max()</code>        | <code>round()</code>      |                             |

# One Function – One Action

- ▶ Functions should be short and do exactly one thing
- ▶ Two independent actions usually deserve two functions, even if they are usually called together (in that case we can make a 3rd function that calls those two)
- ▶ A separate function is not only easier to test and debug – its very existence is a great comment!
- ▶ A few examples of breaking this rule:
  - ▶ `get_age` – would be bad if it also prints the age (should only get)
  - ▶ `check_permission` – would be bad if displays the access granted/denied message (should only perform the check and return the result)

# DocStrings

- ▶ Documentation strings (docstrings) are used to describe what a function does
- ▶ The docstring must be the first indented statement in the function or class
- ▶ Triple quotes are used while writing docstring
- ▶ A function docstring is a string literal that occurs as the first statement of the function definition. It should be written as a triple-quoted string on a single line if the function is simple, or on multiple lines with an initial one-line summary for more detailed descriptions of complex functions

```
def solve_quadratic(a, b, c):
 """Return the roots of ax^2 + bx + c."""
 d = b**2 - 4*a*c;
 r1 = (-b + math.sqrt(d)) / (2*a)
 r2 = (-b - math.sqrt(d)) / (2*a)
 return r1, r2
```

# DocStrings

- ▶ The docstring becomes the special `__doc__` attribute of the function:

```
roots.__doc__
```

```
'Return the roots of ax^2 + bx + c.'
```

- ▶ From an interactive shell, typing `help(function_name)` provides more detailed information concerning the function, including this docstring:

```
help(solve_quadratic)
```

```
Help on function solve_quadratic in module __main__:
```

```
solve_quadratic(a, b, c)
 Return the roots of ax^2 + bx + c.
```

# DocStrings

- ▶ We will use Google coding style for writing docstrings:

```
def func(param1, param2):
 """This is an example of Google style.
```

Args:

    param1: This is the first param.  
    param2: This is a second param.

Returns:

    This is a description of what is returned.

Raises:

    KeyError: Raises an exception.

"""

```
def square_root(n):
 """Calculate the square root of a number.
```

Args:

    n: the number to get the square root of.

Returns:

    the square root of n.

Raises:

    TypeError: if n is not a number.

    ValueError: if n is negative.

"""

# DocStrings

## ► Another example for a well-documented function:

```
def fetch_bigtable_rows(big_table, keys, other_silly_variable=None):
 """Fetches rows from a Bigtable.

 Retrieves rows pertaining to the given keys from the Table instance represented by big_table.
 Silly things may happen if other_silly_variable is not None.

 Args:
 big_table: An open Bigtable Table instance.
 keys: A sequence of strings representing the key of each table row to fetch.
 other_silly_variable: Another optional variable, that has a much longer name than
 the other args, and which does nothing.

 Returns:
 A dict mapping keys to the corresponding table row data fetched.
 Each row is represented as a tuple of strings. For example:

 {'Serak': ('Rigel VII', 'Preparer'),
 'Zim': ('Irk', 'Invader'),
 'Lrrr': ('Omicron Persei 8', 'Emperor')}

 If a key from the keys argument is missing from the dictionary,
 then that row was not found in the table.

 Raises:
 IOError: An error occurred accessing the bigtable.Table object.
 """


```

## Exercise (1)

- ▶ Write a function that returns the greatest common divisor (GCD) of two numbers
- ▶ The greatest common divisor of two numbers is the largest positive integer that perfectly divides the two given numbers. For example, the GCD of 12 and 15 is 3.
- ▶ Use the Euclidean algorithm for finding the GCD:
  - ▶ Divide the greater number by the smaller and take the remainder
  - ▶ Now, divide the smaller by this remainder
  - ▶ Repeat until the remainder is 0
- ▶ For example:  $\text{gcd}(1071, 462) = \text{gcd}(462, 1071 \bmod 462)$   
 $= \text{gcd}(462, 147)$   
 $= \text{gcd}(147, 462 \bmod 147)$   
 $= \text{gcd}(147, 21)$   
 $= \text{gcd}(21, 147 \bmod 21)$   
 $= \text{gcd}(21, 0)$   
 $= 21$

## Exercise (2)

- ▶ A **perfect number** is an integer that is equal to the sum of its proper divisors (the sum of its divisors excluding the number itself)
  - ▶ The first perfect number is 6, since  $6 = 1 + 2 + 3$
  - ▶ The next perfect number is 28, since  $28 = 1 + 2 + 4 + 7 + 14$
- ▶ There are many interesting mathematical open problems regarding perfect numbers
  - ▶ For example, it is not known whether there are any odd perfect numbers, nor whether infinitely many perfect numbers exist
- ▶ Write a function `is_perfect(n)` that returns whether  $n$  is a perfect number or not
- ▶ Write a function `find_perfect_numbers(n)` that returns a list of all the perfect numbers which are less than or equal to  $n$
- ▶ Use the last function to print all perfect numbers less than 10,000
- ▶ Use docstrings to document all your functions

## Exercise (3)

- ▶ Write a function rank(scores) that gets a list of scores, and produces a list associating each score with a rank (starting with 1 for the highest score)
- ▶ Equal scores should have the same rank
- ▶ Sample usage of the function:

```
scores = [87, 75, 75, 50, 32, 32]
score_ranks = rank(scores)
score_ranks
```

```
[1, 2, 2, 4, 5, 5]
```

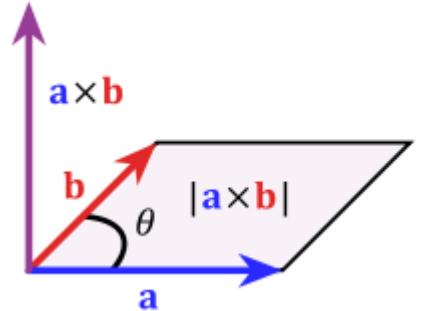
## Exercise (4)

- ▶ Write two functions which, given two lists of length 3 representing three dimensional vectors  $\mathbf{a}$  and  $\mathbf{b}$ , calculate the dot product,  $\mathbf{a} \cdot \mathbf{b}$ , and the vector (cross) product,  $\mathbf{a} \times \mathbf{b}$
- ▶ **Reminder:** the dot product of two vectors  $\mathbf{a}$  and  $\mathbf{b}$  is defined as:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^n a_i b_i = \| \mathbf{a} \| \| \mathbf{b} \| \cos(\theta)$$

- ▶ where  $\theta$  is the angle between  $\mathbf{a}$  and  $\mathbf{b}$
- ▶ The cross product of two vectors  $\mathbf{a}$  and  $\mathbf{b}$  is defined only for 3-dimensional vectors:  

$$\mathbf{a} \times \mathbf{b} = (a_2 b_3 - a_3 b_2, a_3 b_1 - a_1 b_3, a_1 b_2 - a_2 b_1) = \| \mathbf{a} \| \| \mathbf{b} \| \sin(\theta) \mathbf{n}$$
- ▶ where  $\mathbf{n}$  is a unit vector perpendicular to the plane containing  $\mathbf{a}$  and  $\mathbf{b}$  in the direction given by the right-hand rule
- ▶ The cross product is a vector that is perpendicular to both  $\mathbf{a}$  and  $\mathbf{b}$



## Exercise (5)

- ▶ A DNA sequence encodes each amino acid making up a protein as a three nucleotide sequence called a *codon*
- ▶ For example, the sequence fragment AGTCTTATATCT contains the codons (AGT, CTT, ATA, TCT) if read from the first position (“*frame*”)
  - ▶ If read in the second frame it yields the codons (GTC, TTA, TAT), and in the third (TCT, TAT, ATC)
- ▶ Write a function that extracts the codons into a list of 3-letter strings given a DNA sequence and a frame as an integer value (0, 1 or 2)

# Variable Scope

- ▶ The scope of a variable refers to the part of the program where it can be accessed
- ▶ **Local variables** are variables created inside a function
  - ▶ Local variables can only be accessed inside the body of the function in which it is defined
  - ▶ Local variables are subject to garbage collection as soon as the function ends
    - ▶ As a result, trying to access a local variable outside its scope will result in an error
- ▶ **Global variables** are defined outside of any function
  - ▶ The scope of a global variable starts from the point where it is defined and continues until the end of the program
- ▶ Some notes on global variables:
  - ▶ Use global variables only when it's important that these variables are accessible from anywhere
  - ▶ Modern code has few or no globals
  - ▶ Most variables reside in their functions

# Variable Scope

## ▶ Example:

```
global_var = 5 # a global variable

def func():
 local_var = 10 # a local variable, only available inside func()
 print("Inside func(): local_var =", local_var)
 print("Inside func(): global_var =", global_var) # accessing a global variable inside a function

func()
print("Outside func(): global_var =", global_var)

Inside func(): local_var = 10
Inside func(): global_var = 5
Outside func(): global_var = 5
```

## ▶ The global variable can be defined after the function is defined, but must be before the function is called:

```
def func():
 a = 5
 print(a, b)
b = 6
func()
```

5 6

# Variable Scope

- If we have a local and a global variable with the same name, the local variable *shadows* the global one:

```
user_name = "John"
def show_message():
 user_name = "Bob" # shadowing
 print("Hello,", user_name)

show_message()
print(user_name)
```

```
Hello, Bob
John
```

The local variable `user_name` exists only within the body of the function. It disappears after the function exits, and doesn't overwrite the global `user_name`.

- Python's rules for resolving scope can be summarized as **LEGB**:  
Local scope -> Enclosing scope -> Global scope -> Built-ins
- Thus, if you happen to give a variable the same name as a built-in function (such as `range` or `len`), then that name resolves to your variable
  - It is therefore generally not a good idea to name your variables after built-ins

# The global keyword

- ▶ A function cannot *modify* variables defined outside its local scope (“rebind” them to new objects)
- ▶ Since an assignment statement inside a function is interpreted as defining a new local variable

```
def func1():
 print(x) # OK, providing x is defined in global or enclosing scope
```

```
def func2():
 x += 1 # Not OK: can't modify x if it isn't local
```

```
x = 4
func1()
```

```
4
```

```
func2()
```

```

UnboundLocalError Traceback (most recent call last)
<ipython-input-139-1159c30513e1> in <module>()
 1 func2()

<ipython-input-137-aeafdc746753> in func2()
 1 def func2():
 2 x += 1 # Not OK: can't modify x if it isn't local

UnboundLocalError: local variable 'x' referenced before assignment
```

# The global keyword

- ▶ The **global** keyword allows you to modify a global variable within a local scope

```
def func2():
 global x
 x += 1 # OK now - Python knows we mean x in global scope
```

```
x = 4
func2() # No error
x
```

5

- ▶ You should think carefully whether it is really necessary to use this technique
  - ▶ Would it be better to pass x as an argument and return its updated value from the function?
  - ▶ Especially in longer programs, variable names in one scope that change value within functions lead to confusing code, behavior that is hard to predict and tricky bugs

# Nested Functions

- ▶ **Nested functions** are functions defined within other functions
  - ▶ An arbitrary level of nesting is possible
- ▶ We can use nested functions to organize our code, like this:

```
def say_hi_bye(first_name, last_name):
 def get_full_name():
 return first_name + " " + last_name

 print("Hi,", get_full_name())
 print("Bye,", get_full_name())
```

```
say_hi_bye("Adam", "Smith")
```

Hi, Adam Smith  
Bye, Adam Smith

# NonLocal Variables

- ▶ A nested function can read variables declared in its enclosing scope
  - ▶ But cannot assign new values to that variables
- ▶ Similarly to global, a variable can be declared in the nested function as **nonlocal**
- ▶ Once this is done, the nested function can assign a new value to that variable and that modification is going to be seen outside of the nested function:

```
def outer():
 x = 5

 def nested():
 nonlocal x
 x += 1

 nested()
 print(x)

outer()
```

# Closures

- ▶ A **closure** is a nested function that is returned by the outer function, and can access the outer variables even after the outer function has finished its execution
- ▶ Closures can avoid the use of global values and provide some form of data hiding

```
def make_multiplier_of(n):
 def multiplier(x):
 return x * n
 return multiplier
```

```
times3 = make_multiplier_of(3)
times5 = make_multiplier_of(5)
```

```
print(times3(9))
```

27

```
print(times5(3))
```

15

- ▶ Note that even when the outer function has finished its execution, the closure function `multiplier(x)` returned by it can refer to the variable of the outer function (`n` in this case)

# Closures

- ▶ All closure functions have a `__closure__` attribute that returns a tuple of cell objects
- ▶ The cell object has an attribute `cell_contents` which stores the closed value:

```
times3.__closure__[0].cell_contents
```

```
3
```

```
times5.__closure__[0].cell_contents
```

```
5
```

## Exercise (6)

- ▶ Study the following code and predict the result before running it:

```
def outer_func():
 def inner_func():
 a = 9
 print("inside inner_func, a is {} (id={})".format(a, id(a)))
 print("inside inner_func, b is {} (id={})".format(b, id(b)))
 print("inside inner_func, len is {} (id={})".format(len, id(len)))
 len = 2
 print("inside outer_func, a is {} (id={})".format(a, id(a)))
 print("inside outer_func, b is {} (id={})".format(b, id(b)))
 print("inside outer_func, len is {} (id={})".format(len, id(len)))
 inner_func()

a, b = 6, 7
outer_func()
print("in global scope, a is {} (id={})".format(a, id(a)))
print("in global scope, b is {} (id={})".format(b, id(b)))
print("in global scope, len is {} (id={})".format(len, id(len)))
```

# Parameter Passing

- ▶ The two common mechanisms for passing arguments to functions are:
  - ▶ **Call by Value** – a copy of the actual arguments is passed to the respective formal arguments, hence any change to the values of the formal arguments will not affect the variables in the caller's scope.
  - ▶ **Call by Reference** – the location (address) of the actual arguments is passed to the formal arguments, hence any change made to the formal arguments will also reflect in the actual arguments.
- ▶ Python uses a mechanism which is known as "**Call by Object**":
  - ▶ Recall that everything in Python is object
  - ▶ Thus, when a function is called with arguments, the address of the object stored in the argument is passed to the parameter variable
  - ▶ If you pass immutable arguments like strings or tuples, the passing acts like call-by-value: the object reference is passed to the function parameters but they can't be changed in the function.
  - ▶ If you pass mutable arguments (such as lists), they are also passed by object reference, but they can be changed in place in the function.

# Passing Immutable Objects

```
def func1(a):
 print("func1: a = {}, id = {}".format(a, id(a)))
 a = 7 # reassigns local a to the integer 7
 print("func1: a = {}, id = {}".format(a, id(a)))
```

```
a = 3
print("global: a = {}, id = {}".format(a, id(a)))
```

```
global: a = 3, id = 1387949600
```

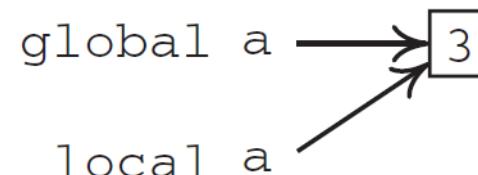
```
func1(a)
```

```
func1: a = 3, id = 1387949600
func1: a = 7, id = 1387949728
```

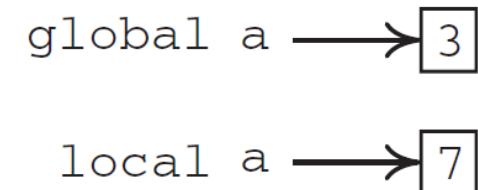
```
print("global: a = {}, id = {}".format(a, id(a)))
```

```
global: a = 3, id = 1387949600
```

Before reassigning the local variable a



After reassigning the local variable a



# Passing Mutable Objects

```
def func2(b):
 print("func1: b = {}, id = {}".format(b, id(b)))
 b.append(7) # add an item to the list
 print("func1: b = {}, id = {}".format(b, id(b)))
```

```
c = [1, 2, 3]
print("global: c = {}, id = {}".format(c, id(c)))
```

```
global: c = [1, 2, 3], id = 2122945792840
```

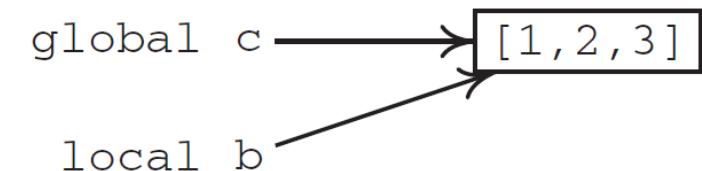
```
func2(c)
```

```
func1: b = [1, 2, 3], id = 2122945792840
func1: b = [1, 2, 3, 7], id = 2122945792840
```

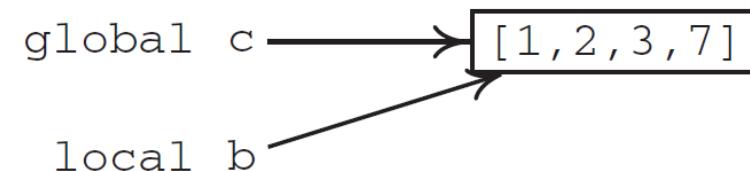
```
print("global: c = {}, id = {}".format(c, id(c)))
```

```
global: c = [1, 2, 3, 7], id = 2122945792840
```

Before appending to the list



After appending to the list



# Keyword Arguments

- ▶ Arguments to functions can be passed in two ways:
  - ▶ **Positional arguments** – the arguments are passed in the same order as their respective parameters in the function header
  - ▶ **Keyword arguments** – the arguments are passed in an arbitrary order by setting them explicitly, using the form: `kwarg=value`
- ▶ Here are some different ways to call `solve_quadratic()` using keyword arguments:

```
solve_quadratic(a=1, c=-6, b=-1)
```

```
(3.0, -2.0)
```

```
solve_quadratic(b=-1, a=1, c=-6)
```

```
(3.0, -2.0)
```

```
solve_quadratic(c=-6, b=-1, a=1)
```

```
(3.0, -2.0)
```

# Keyword Arguments

- ▶ Keyword arguments are useful in the following scenarios:
  - ▶ You want to call arguments by their names to make it more clear what they represent
  - ▶ You want to rearrange arguments in a way that makes them most readable
  - ▶ You want to leave out arguments that have default values (to be discussed next)
- ▶ We can also mix positional and keyword arguments in a function call
- ▶ In such case the positional arguments must come before any keyword arguments
  - ▶ Otherwise Python won't know to which variable the positional argument corresponds

```
solve_quadratic(1, c=-6, b=-1) # OK
```

```
(3.0, -2.0)
```

```
solve_quadratic(b=-1, 1, -6) # Oops: which is a and which is c?
```

```
File "<ipython-input-58-a11319e0cccd7>", line 1
 solve_quadratic(b=-1, 1, -6) # Oops: which is a and which is c?
^
```

```
SyntaxError: positional argument follows keyword argument
```

# Default Arguments

- ▶ Sometimes you want to define a function that takes an *optional argument*: if the caller doesn't provide a value for this argument, a default value is used
- ▶ Default arguments are set in the function definition, by using an assignment operator following the parameter name:

```
def log(message, severity="INFO"):
 print("{}: {}".format(severity, message))
```

```
log("File closed")
```

```
INFO: File closed
```

```
log("Cannot open file", "ERROR")
```

```
ERROR: Cannot open file
```

- ▶ Parameters with default arguments must be the trailing parameters in the function declaration parameter list

# Default Arguments

- In a function call, when you skip some parameters with default arguments, and want to provide values for the rest, you must use keyword arguments:

```
def log(message, severity="INFO", source="FileSystem"):
 print("[{} {}]: {}".format(source, severity, message))
```

```
log("Table created", source="SQLServer")
```

```
[SQLServer] INFO: Table created
```

# Default Arguments

- ▶ Default arguments are assigned when the Python interpreter first encounters the function definition
- ▶ Therefore, if a function is defined with an argument defaulting to the value of some variable, subsequently changing that variable *will not change the default*:

```
default_severity="INFO"
def log(message, severity=default_severity, source="FileSystem"):
 print("[{} {}]: {}".format(source, severity, message))
```

```
log("File closed")
```

```
[FileSystem] INFO: File closed
```

```
default_severity="WARNING"
log("File closed")
```

```
[FileSystem] INFO: File closed
```

# Mutable Default Arguments

- ▶ This can lead to some unexpected results, particularly for mutable arguments:

```
def append_to(element, alist=[]):
 alist.append(element)
 return alist
```

```
list1 = append_to(12)
print(list1)

list2 = append_to(25)
print(list2)
```

```
[12]
[12, 25]
```

- ▶ A new list is created *once* when the function is defined, and the same list is used in each successive call
- ▶ This means that if you use a mutable default argument and mutate it, you have mutated that object for all future calls to the function as well

# Mutable Default Arguments

- If you want to create a new object each time the function is called, you can use a default arg to signal that no argument was provided (**None** is often a good choice):

```
def append_to(element, alist=None):
 if alist is None:
 alist = []
 alist.append(element)
 return alist
```

```
list1 = append_to(12)
print(list1)

list2 = append_to(25)
print(list2)
```

```
[12]
[25]
```

# Variable Length Argument Lists

- ▶ In Python you can pass an arbitrary number of arguments to functions
- ▶ A single asterisk (`*args`) is used to pass a *non-keyworded*, variable-length argument list
  - ▶ `*args` will be bound to a tuple containing any positional arguments which don't correspond to any formal parameter
- ▶ A double asterisk (`**kargs`) is used to pass a *keyworded*, variable-length argument list
  - ▶ `**kargs` will be bound to a dictionary containing any keyword arguments which don't correspond to any formal parameters
- ▶ The arguments in a function or a function call must appear in the following order:
  - ▶ Positional arguments
  - ▶ `*args`
  - ▶ Keyword arguments
  - ▶ `**kwargs`

# Arbitrary Positional Arguments

- ▶ Examples for functions with arbitrary positional arguments:

```
def multiply(*args):
 result = 1
 for num in args:
 result *= num
 return result
```

```
multiply(1, 2, 3, 4, 5)
```

```
120
```

```
multiply() # args will be an empty tuple
```

```
1
```

```
def test_var_args(f_arg, *args):
 print("first normal arg:", f_arg)
 for arg in args:
 print("another arg through *args:", arg)

test_var_args('yasoob', 'python', 'eggs', 'test')
```

```
first normal arg: yasoob
another arg through *args: python
another arg through *args: eggs
another arg through *args: test
```

- ▶ The built-in function `print()` takes an unlimited number of positional arguments, thus the optional `sep`, `end`, `file`, and `flush` attributes must be passed as keyword arguments:

```
print("comma", "separated", "word", sep=", ")
```

```
comma, separated, word
```

# Arbitrary Keyword Arguments

- ▶ An example for a function with arbitrary keyword arguments:

```
def print_values(**kwargs):
 for key, value in kwargs.items():
 print("The value of {} is {}".format(key, value))

print_values(my_name="thor", your_name="hulk")
```

The value of my\_name is thor  
The value of your\_name is hulk

# Arbitrary Keyword Arguments

- ▶ Combining arbitrary keyword arguments with arbitrary positional arguments:

```
def print_receipt(order_id, *food_list, **keywords):
 print("Order #", order_id, sep="")
 print("-" * 30)
 for food in food_list:
 print(food)
 print()
 for kw in keywords:
 print(kw, ":", keywords[kw], sep="")
```

```
print_receipt(186, "Greek salad", "Tarragon Chicken", "Creamy Burrito",
 Total=81.5,
 Cash=100,
 Change=18.5)
```

```
Order #186

Greek salad
Tarragon Chicken
Creamy Burrito

Total: 81.5
Cash: 100
Change: 18.5
```

# Keyword-Only Arguments

- ▶ Keyword-only parameters can be defined by including a bare \* in the parameter list of the function definition before them
  - ▶ This ensures that users of the function cannot accidentally use an option that is controlled by a keyword-only argument
- ▶ For example:

```
def func(parameter, *, option1=False, option2=''): pass
```

- ▶ In this example, option1, and option2 are only specifiable via keyword arguments:

```
func(3, option1=True, option2="Hello world")
```

```
func(3, True, "Hello world")
```

```

TypeError Traceback (most recent call last)
<ipython-input-36-7e457df80701> in <module>()
----> 1 func(3, True, "Hello world")
```

```
TypeError: func() takes 1 positional argument but 3 were given
```

# Unpacking Argument Lists

- ▶ \* and \*\* can be used not only in function definitions, but also when calling a function
- ▶ For example, when the arguments are already in a list or tuple, but need to be unpacked for a function call requiring separate positional arguments:

```
def f(x, y, z):
 print("x:", x)
 print("y:", y)
 print("z:", z)
```

```
p = [47, 11, 12]
f(*p)
```

x: 47  
y: 11  
z: 12

```
args = [3, 10]
list(range(*args))
```

[3, 4, 5, 6, 7, 8, 9]

# Unpacking Argument Lists

- In the same fashion, dictionaries can deliver keyword arguments with the `**`-operator:

```
def parrot(voltage, state='a stiff', action='voom'):
 print("-- This parrot wouldn't", action, end=' ')
 print("if you put", voltage, "volts through it.", end=' ')
 print("E's", state, "!")

d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
parrot(**d)
```

```
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

## Exercise (7)

- Given the definition of the following six functions:

```
def f1(a, b): print(a, b)
def f2(a, *b): print(a, b)
def f3(a, **b): print(a, b)
def f4(a, *b, **c): print(a, b, c)
def f5(a, b=2, c=3): print(a, b, c)
def f6(a, b=2, *c): print(a, b, c)
```

- What will be the result of the following expressions? Explain and test each result

```
f1(1, 2)
f1(b=2, a=1)
f2(1, 2, 3)
f3(1, x=2, y=3)
f4(1, 2, 3, x=2, y=3)
f5(1)
f5(1, 4)
f6(1)
f6(1, 3, 4)
```

# Lambda Functions

- ▶ A lambda function in Python is a type of *anonymous* (unnamed) function
- ▶ It is used primarily to write short functions that are a hassle to define as regular functions
- ▶ A lambda function has the following syntax:

```
lambda arguments: expression
```

- ▶ Lambda functions can have any number of arguments but only one expression
- ▶ The expression is evaluated and returned
- ▶ The expression cannot contain statements such as loop blocks, conditionals or print statements
- ▶ Lambda functions can be used wherever function objects are required

# Lambda Functions

- ▶ An example for a lambda function that doubles the input value:

```
double = lambda x: x * 2

print(double(5))
```

10

- ▶ The argument 5 is passed to x and the result of the expression specified in the lambda definition after the colon is passed back to the caller
- ▶ To pass more than one argument to a lambda function, pass a tuple (without parentheses):

```
f = lambda x, y: (x + y) ** 2
f(2, 3)
```

25

# Lambda Functions as Arguments

- ▶ Lambda is often used as an argument to other functions that expect a function object
- ▶ For example, the built-in sorted() and sort() functions can order lists based on a **key function**, which is called on each element prior to making comparisons
- ▶ For example, sorting a list of strings is case sensitive by default:

```
sorted("Nobody expects the Spanish Inquisition".split())
```

```
['Inquisition', 'Nobody', 'Spanish', 'expects', 'the']
```

- ▶ We can make the sorting case insensitive, however, by passing each word to the str.lower (or str.upper) method:

```
sorted("Nobody expects the Spanish Inquisition".split(), key=str.lower)
```

```
['expects', 'Inquisition', 'Nobody', 'Spanish', 'the']
```

- ▶ We do not use parentheses here, as in str.lower(), because we are passing the *function itself* to the key argument, not calling it directly

# Lambda Functions as Arguments

- ▶ It is typical to use lambda expressions to provide simple anonymous functions for other functions
- ▶ For example, the built-in function **sorted(iterable, \*, key=None, reverse=False)** has an optional parameter **key**, that specifies a function of one argument that is used to extract a comparison key from each list element
- ▶ In the following example we use a lambda expression to sort a list of tuples by the second item of each tuple, so the noble gases are sorted by their atomic number:

```
noble_gases = [("argon", 18), ("neon", 10), ("xenon", 54),
 ("krypton", 36), ("helium", 2), ("radon", 86)]
sorted(noble_gases, key=lambda e: e[1])

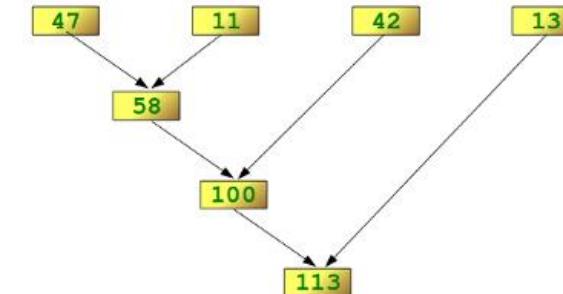
[('helium', 2),
 ('neon', 10),
 ('argon', 18),
 ('krypton', 36),
 ('xenon', 54),
 ('radon', 86)]
```

# Reducing a List

- ▶ The function **reduce(func, seq)** continually applies the function **func()** to the sequence **seq**, until a single value is returned
- ▶ If **seq = [ s<sub>1</sub>, s<sub>2</sub>, s<sub>3</sub>, ... , s<sub>n</sub> ]**, calling **reduce(func, seq)** works like this:
  - ▶ At first, the first two elements of seq will be applied to func, i.e. **func(s<sub>1</sub>, s<sub>2</sub>)**
  - ▶ The list on which reduce() works looks now like this: **[ func(s<sub>1</sub>, s<sub>2</sub>), s<sub>3</sub>, ... , s<sub>n</sub> ]**
  - ▶ Next, func() is applied on the previous result and the third element, i.e. **func(func(s<sub>1</sub>, s<sub>2</sub>), s<sub>3</sub>)**
  - ▶ The list looks like this now: **[ func(func(s<sub>1</sub>, s<sub>2</sub>), s<sub>3</sub>), ... , s<sub>n</sub> ]**
  - ▶ Continue like this until just one element is left and return this element as the result of reduce()

```
from functools import reduce
reduce(lambda x, y: x + y, [47, 11, 42, 13])
```

113



## Exercise (8)

- ▶ Define a function **apply(func, list)** that gets a function and a list, and applies the function on every item in the list (modifying the list in place)
- ▶ Use `apply()` to multiply by 5 all the numbers in a given list of numbers
- ▶ Use `apply()` to capitalize all the words in a given list (i.e., change the first letter to uppercase)

## Exercise (9)

- Predict the result of the following program before running it:

```
def func1(f):
 return lambda x: f(x + 1)

func2 = func1(lambda x: x * x)
func2(2)
```

# Generators

- ▶ A *generator* is a function that returns a *generator iterator* by calling **yield**
- ▶ It is a function that behaves like an iterable object
  - ▶ i.e., a function that can be used in a for loop and that will yield its values, in turn, on demand
  - ▶ This is often more efficient than calculating and storing all of the values that will be iterated over
- ▶ A generator function looks just like a regular Python function, but instead of exiting with a return value, it contains a **yield** statement which returns a value each time it is required to by the iteration
- ▶ In fact, we've been using generators already because the familiar `range()` built-in function is a type of generator object (from Python 3)

# Generators

- ▶ As a simple example, let's define a generator, **count()**, to count to n:

```
def count(n):
 for i in range(n):
 yield i
```

```
for j in count(5):
 print(j)
```

```
0
1
2
3
4
```

- ▶ Note that we can't simply call our generator like a regular function:

```
count(5)
<generator object count at 0x0000019E19DD6678>
```

- ▶ The generator count is expecting to be called as part of a loop and on each iteration it yields its result and stores its state (the value of i) until the loop next calls upon it

# Generator Comprehension

- ▶ There is a *generator comprehension* syntax similar to list comprehension (use round brackets instead of square brackets):

```
squares = (x**2 for x in range(5))
```

```
for square in squares:
 print(square)
```

```
0
1
4
9
16
```

- ▶ However, once we have “exhausted” our generator comprehension, we cannot iterate over it again without redefining it:

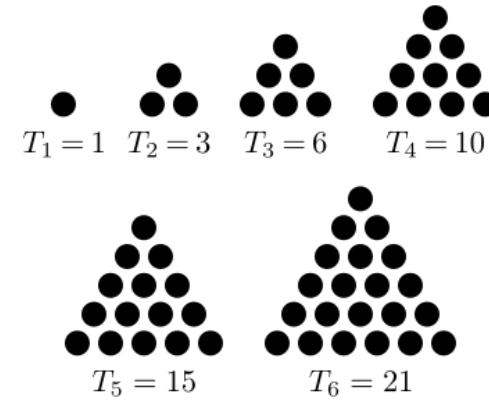
```
for square in squares:
 print(square)
```

- ▶ To obtain a list or tuple of a generator’s values, simply pass it to list() or tuple()

## Exercise (10)

- ▶ Define a generator `triangular_numbers(n)` that generates the triangular numbers:

$$T_n = \sum_{k=1}^n k = 1+2+3+\dots+n \quad n = 0, 1, 2, \dots$$



- ▶ Use this function to generate the list of the first 15 triangular numbers

# Filtering

- ▶ The built-in function **filter(func, iter)** returns a generator-like object, that generates the elements of an iterable, for which the function *func* returns True
- ▶ In the following example, the odd integers less than 10 are generated:

```
list(filter(lambda x: x % 2 == 1, range(10)))
```

```
[1, 3, 5, 7, 9]
```

- ▶ This statement is equivalent to the list comprehension:

```
[x for x in range(10) if x % 2 == 1]
```

```
[1, 3, 5, 7, 9]
```

# The map() Function

- ▶ The built-in function **map(func, iter)** returns a generator-like object that applies the function *func* to all the elements of the iterable *iter*, yielding the results one by one
- ▶ For example, one way to sum a list of lists is to map the sum built-in to it:

```
mylists = [[1, 2, 3], [10, 20, 30], [25, 75, 100]]
list(map(sum, mylists))
```

```
[6, 60, 200]
```

- ▶ This statement is equivalent to the list comprehension:

```
[sum(l) for l in mylists]
```

```
[6, 60, 200]
```

## Exercise (11)

- ▶ You are given the following sentence:

```
s = ["There", "are", "no", "miracles", "in", "this", "world"]
```

- ▶ Print the lengths of all the words in the sentence that contain the letter 'a'
  - ▶ Using map() and filter()
  - ▶ Using list comprehension

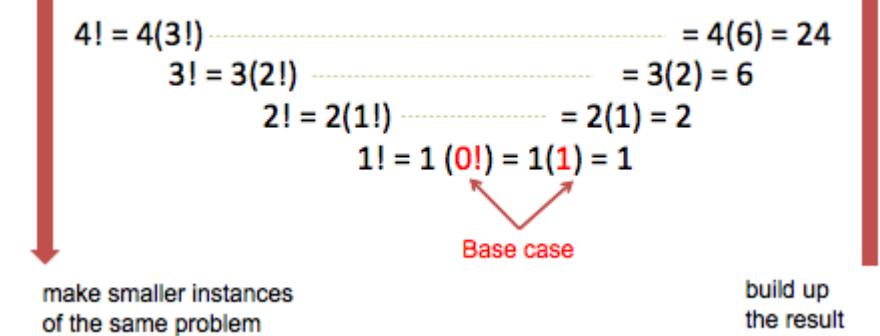
# Recursion

- ▶ A **recursive function** is a function that calls itself
- ▶ Recursion is not always necessary but can often lead to elegant algorithms
- ▶ Following is an example of recursive function to find the factorial of an integer
  - ▶ The algorithm makes use of the fact that  $n! = n \cdot (n-1)!$

```
def factorial(n):
 """A recursive function to find the factorial of an integer"""
 if n == 1:
 return 1
 else:
 return n * factorial(n - 1)
```

```
num = int(input("Enter a number:"))
print("The factorial of", num , "is", factorial(num))
```

```
Enter a number:5
The factorial of 5 is 120
```



# Recursion

---

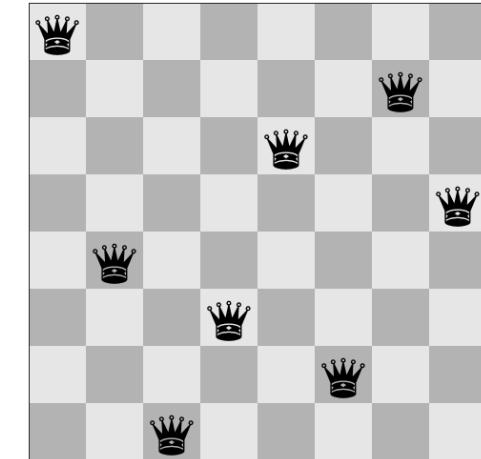
- ▶ Advantages of recursion:
  - ▶ Recursive functions make the code look clean and elegant
  - ▶ A complex task can be broken down into simpler sub-problems using recursion
  - ▶ Sequence generation is easier with recursion than using some nested iterations
- ▶ Disadvantages of recursion:
  - ▶ Sometimes the logic behind recursion is hard to follow through
  - ▶ Recursive calls are expensive (inefficient) as they take up a lot of memory and time
  - ▶ Recursive functions are hard to debug

## Exercise (12)

- ▶ Write a function that determines if a string is a palindrome (that is, reads the same backward as forward) **using recursion**
- ▶ Some examples of palindromic words are  
*redivider, deified, civic, radar, level, rotor, kayak, reviver, racecar, redder, madam, and refer*

## Exercise (13)

- ▶ The **N-Queen problem** is the problem of placing N chess queens on an NxN chessboard, so that no two queens attack each other
- ▶ Thus, a solution requires that no two queens share the same row, column, or diagonal
- ▶ For example, the following is a solution for 8-Queen problem:



- ▶ Write a function that gets the board size N, prints the number of possible placements of N queens on the board of size NxN, and all the possible boards

# Exercise (13) – Cont.

## ▶ Example:

```
count = 0
place_queens(6)
print("Number of boards:", count)
```

```
.Q....
...Q..
....Q
Q.....
..Q...
....Q.
```

```
..Q...
....Q
.Q...
....Q.
Q.....
...Q..
```

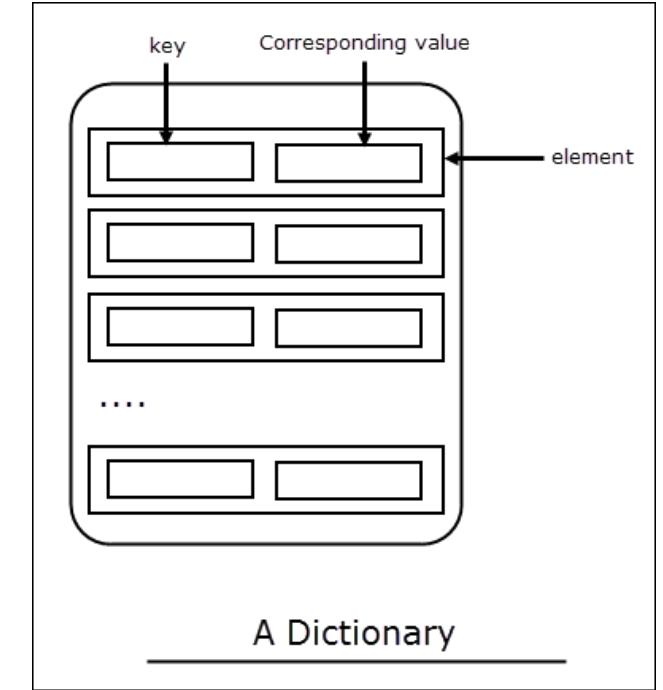
```
...Q..
Q....
....Q.
.Q...
....Q
..Q...
```

```
....Q.
..Q...
Q.....
....Q
...Q..
.Q....
```

```
Number of boards: 4
```

# Dictionaries

- ▶ A **dictionary** is a collection of key-value pairs
  - ▶ Also known as “associative array” or “hash” in other languages
- ▶ Unlike sequences such as lists and tuples, in which items are indexed by an integer, each item in a dictionary is indexed by a unique key
- ▶ The items are stored in the dictionary in no particular order
- ▶ The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings or tuples
  - ▶ Actually, dictionary keys can be any hashable objects, i.e. objects with a `__hash__()` method that generates a particular integer from any instance of that object
- ▶ Dictionaries themselves are mutable objects



# Defining a Dictionary

- ▶ A dictionary is defined by giving key: value pairs between braces:

```
variable_name = {
 'key1': value1,
 'key2': value2,
 ...
 'keyN': valueN
}
```

- ▶ For example:

```
height = {
 'Burj Khalifa': 829.8,
 'One World Trade Center': 541.3,
 'Tokyo Skytree': 634,
 '432 Park Avenue': 425.5,
 'KVLY-TV Mast': 628.8
}
```

- ▶ Although the dictionary keys must be unique, the dictionary values need not be

# Defining a Dictionary

- ▶ The command **print()** will display the dictionary in the same format (between braces), but in no particular order

```
print(height)
```

```
{'Burj Khalifa': 829.8, 'One World Trade Center': 541.3, 'Tokyo Skytree':
634, '432 Park Avenue': 425.5, 'KVLY-TV Mast': 628.8}
```

- ▶ We can also use **dict()** constructor function to define a dictionary
  - ▶ The function requires you to pass a sequence of (key, value) pairs
  - ▶ If the keys are simple strings (i.e., strings that could be used as variable names), the pairs can also be specified as keyword arguments to the constructor:

```
capitals = dict([('US', 'Washington'), ('France', 'Paris'), ('UK', 'London')])
mass = dict(Mercury=3.301e23, Venus=4.867e24, Earth=5.972e24)
```

- ▶ To create an empty dictionary use {}, or dict()

# Accessing Values in a Dictionary

- An individual item can be retrieved by indexing it with its key, either as a literal or with a variable equal to the key:

```
height['One World Trade Center']
```

541.3

```
building = 'Tokyo Skytree'
height[building]
```

634

- If the specified key doesn't exist, a `KeyError` exception will be raised:

```
height['The Shard']
```

```

KeyError Traceback (most recent call last)
<ipython-input-30-3c9dcfb9ffab> in <module>()
----> 1 height['The Shard']
```

```
KeyError: 'The Shard'
```

# Adding and Modifying Values

- ▶ Items in a dictionary can also be *assigned* by indexing it in this way:

```
height['Empire State Building'] = 381
height

{'432 Park Avenue': 425.5,
 'Burj Khalifa': 829.8,
 'Empire State Building': 381,
 'KVLY-TV Mast': 628.8,
 'One World Trade Center': 541.3,
 'Tokyo Skytree': 634}
```

- ▶ If the key already exists in the dictionary, then its value is updated:

```
height['Empire State Building'] = 443 # with antenna included
height

{'432 Park Avenue': 425.5,
 'Burj Khalifa': 829.8,
 'Empire State Building': 443,
 'KVLY-TV Mast': 628.8,
 'One World Trade Center': 541.3,
 'Tokyo Skytree': 634}
```

# Deleting Elements

- ▶ To delete an element from a dictionary, use the **del** statement:

```
del height['KVLY-TV Mast']
height

{'432 Park Avenue': 425.5,
 'Burj Khalifa': 829.8,
 'Empire State Building': 443,
 'One World Trade Center': 541.3,
 'Tokyo Skytree': 634}
```

- ▶ If the key doesn't exist in the dictionary, a `KeyError` exception is raised
- ▶ We can use the built-in **len()** function to get the number of elements in a dictionary:

```
len(height)
```

5

# For Loop Iteration

- ▶ We can use a for loop to iterate over all the keys in the dictionary:

```
for building in height:
 print(building, height[building], sep=': ')
```

```
Burj Khalifa: 829.8
One World Trade Center: 541.3
Tokyo Skytree: 634
432 Park Avenue: 425.5
Empire State Building: 443
```

# Membership Operators

- ▶ The **in** and **not in** operators can be used to test the existence of a key inside a dictionary:

```
'Tokyo Skytree' in height
```

True

```
'The Shard' in height
```

False

```
'The Shard' not in height
```

True

# Comparison Operators

- ▶ We can use == and != operators to test whether two dictionaries contain the same elements or not:

```
marks1 = {'Larry': 90, 'Bill': 60}
marks2 = {'Bill': 60, 'Larry': 90}
marks1 == marks2
```

True

- ▶ The other comparison operators such as <, <=, > and >= can't be used with dictionaries because elements in dictionary are stored in no particular order

# Dictionary Methods

- The following table lists some common methods we can use on a dictionary object:

| Method              | Description                                                                                                                                                                                                                                                            |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| keys()              | Returns a sequence containing only the keys from the dictionary                                                                                                                                                                                                        |
| values()            | Returns a sequence containing only the values from the dictionary                                                                                                                                                                                                      |
| items()             | Returns a sequence of tuples, where each tuple contains a key and value of an element                                                                                                                                                                                  |
| get(key, [default]) | Returns the value associated with <i>key</i> . If <i>key</i> is not found it returns None.<br>We can also provide a optional <i>default</i> value as the second argument in which case if the key is not found, <i>default</i> value will be returned instead of None. |
| pop(key)            | Returns the value associated with <i>key</i> and removes the specified key and its corresponding value from the dictionary. If <i>key</i> doesn't exists KeyError exception is raised.                                                                                 |
| popitem()           | Removes and returns a random element from the dictionary as a tuple                                                                                                                                                                                                    |
| copy()              | Creates a new copy of the dictionary                                                                                                                                                                                                                                   |
| clear()             | Removes all elements from the dictionary                                                                                                                                                                                                                               |

# get()

- ▶ Indexing a dictionary with a key that does not exist is an error
- ▶ However, the useful method **get()** can be used to retrieve the value, given a key if it exists, or some default value if it does not
  - ▶ If no default is specified, then None is returned
- ▶ For example:

```
print(mass.get('Pluto'))
```

None

```
mass.get('Pluto', -1)
```

-1

# keys, values and items

- ▶ The three methods: **keys()**, **values()** and **items()**, return respectively, a dictionary's keys, values and key-value pairs (as tuples)
- ▶ In previous versions of Python, each of these methods returned a list, containing a copy of the keys or values, which for most purposes is a wasteful of memory
- ▶ In Python 3 these methods return iterable objects
  - ▶ This is faster and saves memory

```
planets = mass.keys()
print(planets)

dict_keys(['Mercury', 'Venus', 'Earth'])
```

```
for planet in planets:
 print(planet, mass[planet])
```

```
Mercury 3.301e+23
Venus 4.867e+24
Earth 5.972e+24
```

# keys, values and items

- ▶ A `dict_keys` object can be iterated over any number of times, but it is not a list and cannot be indexed or assigned:

```
planets[0]
```

```

TypeError Traceback (most recent call last)
<ipython-input-57-7c16c1f6b877> in <module>()
 1 planets[0]

TypeError: 'dict_keys' object does not support indexing
```

- ▶ If you really want a list of the dictionary's keys, simply pass the `dict_keys` object to the `list()` constructor (which takes any kind of sequence and makes a list out of it):

```
planet_list[1] = 'Jupiter' # doesn't alter the original dictionary's keys
planet_list

['Mercury', 'Jupiter', 'Earth']
```

# keys, values and items

- ▶ Similar methods exist for retrieving a dictionary's values and items (key-value pairs)
  - ▶ The objects returned are dict\_values and dict\_items

```
mass.values()

dict_values([3.301e+23, 4.867e+24, 5.972e+24])
```

```
mass.items()

dict_items([('Mercury', 3.301e+23), ('Venus', 4.867e+24), ('Earth', 5.972e+24)])
```

- ▶ The items() method returns a sequence of tuples, where each tuple contains a key and a value of an element. We can use a for loop to loop over the tuples as follows:

```
for planet, mass in mass.items():
 print(planet, mass, sep=' : ')
```

```
Mercury: 3.301e+23
Venus: 4.867e+24
Earth: 5.972e+24
```

# Dictionary Comprehension

- ▶ A dictionary comprehension is a construct for creating a dictionary based on another iterable object
- ▶ The syntax of dictionary comprehension is:

```
d = {key: value for (key, value) in iterable}
```

- ▶ Each element in the iterable must be iterable itself of two elements
- ▶ For example:

```
keys = ['a', 'b', 'c', 'd', 'e']
values = [1, 2, 3, 4, 5]

my_dict = {k: v for (k, v) in zip(keys, values)}
my_dict

{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
```

- ▶ It seems require more code than just doing this:

```
dict(zip(keys, values))

{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
```

# Dictionary Comprehension

- ▶ However, there are more cases when we can utilize the dictionary comprehension
- ▶ For instance, we can construct a dictionary from a list using comprehension:

```
{x: x**2 for x in [1, 2, 3, 4, 5]}
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

```
{x.upper(): x * 3 for x in 'abcd'}
```

```
{'A': 'aaa', 'B': 'bbb', 'C': 'ccc', 'D': 'ddd'}
```

## Exercise (14)

- ▶ Define a function **count\_words(text)** that gets a text, and prints the list of words in the text with the number of occurrences of each word
- ▶ The words should be printed from the most common word to the least common word
- ▶ *Hint:* use Python's string methods to strip out any punctuation
  - ▶ It suffices to replace any of the following characters with the empty string: !?"';,()'\*.[]
- ▶ Example:

```
text = """ "But I don't want to go among mad people," Alice remarked.
"Oh, you can't help that," said the Cat: "we're all mad here. I'm mad. You're mad."
"How do you know I'm mad?" said Alice.
"You must be," said the Cat, "or you wouldn't have come here." """

print(count_words(text))
```

```
mad: 5, you: 4, said: 3, alice: 2, the: 2, cat: 2, here: 2, im: 2, but: 1, i: 1, dont: 1, want: 1, to:
1, go: 1, among: 1, people: 1, remarked: 1, oh: 1, cant: 1, help: 1, that: 1, were: 1, all: 1, youre:
1, how: 1, do: 1, know: 1, must: 1, be: 1, or: 1, wouldnt: 1, have: 1, come: 1, None
```

## Exercise (15)

- ▶ Anagrams are words that have the same number of same letters, but in different order. For instance:
  - ▶ nap - pan
  - ▶ ear - are - era
  - ▶ cheaters - hectares – teachers
- ▶ Write a function **clean\_anagrams(words)** that returns a list of words cleaned from anagrams
- ▶ For instance:

```
words = ["nap", "teachers", "cheaters", "PAN", "ear", "era", "hectares"]
print(clean_anagrams(words))

['PAN', 'hectares', 'era']
```

# Sets

---

- ▶ A set is an **unordered collection of unique items**
- ▶ As with dictionary keys, elements of a set must be hashable objects
- ▶ A set is useful for removing duplicates from a sequence and for determining the union, intersection and difference between two collections
- ▶ Because they are unordered, set objects cannot be indexed or sliced, but they can be iterated over, tested for membership, and they support the len built-in

# Creating Sets

- ▶ A set is created by listing its elements between braces {...}

```
s = { 1, 1, 4, 3, 2, 2, 3, 1, "surprise!"}
s

{1, 2, 3, 4, 'surprise!'}
```

- ▶ We can also create sets by passing an iterable to the set() constructor:

```
s2 = set([77, 23, 91, 13]) # create a set from a list
s2

{13, 23, 77, 91}
```

```
s3 = set("abc123") # create a set from a string
s3

{'1', '2', '3', 'a', 'b', 'c'}
```

```
s4 = set(("alpha", "beta", "gamma")) # create a set from a tuple
s4

{'alpha', 'beta', 'gamma'}
```

# Sets Built-in Functions

- As with list and tuples, we can also use the following functions with sets:

| Function              | Description                                               |
|-----------------------|-----------------------------------------------------------|
| <code>len(set)</code> | Returns the number of elements in <i>set</i>              |
| <code>sum(set)</code> | Returns the sum of elements in the <i>set</i>             |
| <code>max(set)</code> | Returns the element with the greatest value in <i>set</i> |
| <code>min(set)</code> | Returns the element with the smallest value in <i>set</i> |

```
s1 = {33, 11, 88, 55}
len(s1)
```

4

```
max(s1)
```

88

```
min(s1)
```

11

```
sum(s1)
```

187

# Adding Elements

- ▶ The set method **add()** is used to add elements to the set:

```
s = {2, -2, 0}
s.add(1)
s.add(-1)
s.add(1.0)
s
{-2, -1, 0, 1, 2}
```

- ▶ The last statement doesn't add a new member to the set, since the test `1 == 1.0` is True, so `1.0` is considered to be already in the set
- ▶ We can also add multiple elements to the set at once using the **update()** method, which accepts an iterable object:

```
s.update([2, 3, 4])
s
{-2, -1, 0, 1, 2, 3, 4}
```

# Removing Elements

- ▶ To remove elements there are several methods:
  - ▶ **remove()** removes a specified element but raises a `KeyError` exception if the element is not present in the set
  - ▶ **discard()** does the same, but does not raise an error if the element is not present in the set
  - ▶ **pop** (with no argument) removes an arbitrary element from the set
  - ▶ **clear()** removes all elements from the set

```
s.remove(1)
```

```
s
```

```
{-2, -1, 0, 2, 3, 4}
```

```
s.discard(5) # OK - does nothing
```

```
s
```

```
{-2, -1, 0, 2, 3, 4}
```

```
s.pop()
```

```
0
```

```
s
```

```
{-2, -1, 2, 3, 4}
```

```
s.clear()
```

```
s
```

```
set()
```

# Looping Through Sets

- ▶ Just as with other sequence types, we can use for loop to iterate over the elements of a set:

```
s = {99, 33, 44, 25, 124}
for num in s:
 print(num, end=" ")
```

33 99 44 25 124

# Membership Operators

- ▶ As usual, we can use **in** and **not in** operators to find the existence of an element inside a set:

```
s = {99, 33, 44, 25, 124}
33 in s
```

True

```
5 in s
```

False

```
5 not in s
```

True

# Set Methods

- ▶ Set objects have a wide range of methods related to properties of mathematical sets:

| Method                                                                         | Description                                                                                                                                     |
|--------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>isdisjoint(other)</code><br><br><code>set &lt; other</code>              | Is set disjoint with <i>other</i> ?, i.e., do they have no elements in common?                                                                  |
| <code>issubset(other)</code><br><br><code>set &lt;= other</code>               | Is <i>set</i> a subset of <i>other</i> ?, i.e., are all the elements of <i>set</i> also elements of <i>other</i> ?                              |
| <code>set &lt; other</code>                                                    | Is <i>set</i> a proper subset of <i>other</i> ?, i.e., is <i>set</i> a subset of <i>other</i> but not equal to it?                              |
| <code>issuperset(other)</code><br><br><code>set &gt;= other</code>             | Is <i>set</i> a superset of <i>other</i> ?                                                                                                      |
| <code>set &gt; other</code>                                                    | Is <i>set</i> a proper superset of <i>other</i> ?                                                                                               |
| <code>union(other)</code><br><br><code>set   other   ...</code>                | The union of <i>set</i> and <i>other(s)</i> , i.e., the set of all elements from both <i>set</i> and <i>other(s)</i>                            |
| <code>intersection(other)</code><br><br><code>set &amp; other &amp; ...</code> | The intersection of <i>set</i> and <i>other(s)</i> , i.e., the set of all elements that <i>set</i> and <i>other(s)</i> have in common           |
| <code>difference(other)</code><br><br><code>set - other - ...</code>           | The difference of <i>set</i> and <i>other(s)</i> , i.e., the set of elements in <i>set</i> but not in <i>other(s)</i>                           |
| <code>symmetric_difference(other)</code><br><br><code>set ^ other ^ ...</code> | The symmetric difference of <i>set</i> and <i>other(s)</i> , i.e., the set of elements in either <i>set</i> and <i>other(s)</i> but not in both |

# Set Methods

- ▶ There are two forms for most set expressions: the operator-like syntax requires all arguments to be set objects, whereas explicit method calls will convert any iterable argument into a set

```
A = {1, 2, 3}
B = {1, 2, 3, 4}
A <= B
```

True

```
A.issubset([1, 2, 3, 4]) # OK: [1,2,3,4] is turned into set
```

True

# Set Methods

## ▶ Some more examples:

```
A = {1, 2, 3}
B = {1, 2, 3, 4}
C = {3, 4, 5, 6}
D = {7, 8, 9}
```

```
B | C # union
```

```
{1, 2, 3, 4, 5, 6}
```

```
A | C | D # union of three sets
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
A & C # intersection
```

```
{3}
```

```
C & D
```

```
set()
```

```
C.isdisjoint(D)
```

```
True
```

```
B - C # difference
```

```
{1, 2}
```

```
B ^ C # symmetric difference
```

```
{1, 2, 5, 6}
```

# Frozen Sets

- sets are mutable objects, thus they are *unhashable* and so cannot be used as dictionary keys or as members of other sets

```
a = {1, 2, 3}
b = {'q', (1, 2), a}
```

```

TypeError Traceback (most recent call last)
<ipython-input-39-39b261a243c9> in <module>()
 1 a = {1, 2, 3}
----> 2 b = {'q', (1, 2), a}

TypeError: unhashable type: 'set'
```

- There is, however, a **frozenset** object which is a kind of immutable (and hashable) set
- Frozensets are fixed, unordered collections of unique objects and *can* be used as dictionary keys and set members
  - In a sense, they are to sets what tuples are to lists

# Frozen Sets

- ▶ For example:

```
a = frozenset((1, 2, 3))
b = {'q', (1, 2), a} #OK: the frozen set is hashable
b
```

```
((1, 2), frozenset({1, 2, 3}), 'q')
```

```
b.add(4) # OK: b is a regular set
```

```
a.add(4) # Not OK: frozensets are immutable
```

```

AttributeError Traceback (most recent call last)
<ipython-input-45-209e4590c6b4> in <module>()
----> 1 a.add(4) # Not OK: frozensets are immutable

AttributeError: 'frozenset' object has no attribute 'add'
```

## Exercise (16)

- ▶ Write a function, using set objects, to remove duplicates from a list
- ▶ For example:

```
remove_duplicates([5,7,4,4,5,4,2,8,3,8])
```

```
[2, 3, 4, 5, 7, 8]
```

## Exercise (17)

- ▶ A *Mersenne prime*,  $M_i$ , is a prime number of the form  $M_i = 2^i - 1$
- ▶ Mersenne primes are noteworthy due to their connection to perfect numbers
  - ▶ All even perfect numbers have the form  $2^{p-1}(2^p - 1)$  where  $2^p - 1$  is prime
- ▶ The set of Mersenne primes less than or equal to  $n$  may be thought of as the intersection of the set of all primes less than or equal to  $n$ ,  $P_n$ , with the set,  $A_n$ , of integers satisfying  $2^i - 1 \leq n$
- ▶ Write a program that prints the list of the Mersenne primes less than 1,000,000
- ▶ Hint: Use the [Sieve of Eratosthenes](#) method to find all primes less than or equal to  $n$ ,

## Exercise (18)

- ▶ The *power set* of a set  $S$ ,  $P(S)$ , is the set of all subsets of  $S$ , including the empty set and  $S$  itself
- ▶ For example,  $P(\{1,3,6\}) = \{\{\}, \{1\}, \{3\}, \{6\}, \{1,3\}, \{1,6\}, \{3,6\}, \{1,3,6\}\}$
- ▶ Write a generator function that returns the power set of a given set
- ▶ Usage example:

```
my_set = set([1, 3, 6])
```

```
for s in power_set(my_set):
 print(s)
```

```
set()
{1}
{3}
{1, 3}
{6}
{1, 6}
{3, 6}
{1, 3, 6}
```

# File Handling

- ▶ Until now, data has been hard-coded into our Python programs, and output has been to the console (or the notebook)
- ▶ Of course, it will frequently be necessary to input data from an external file and to write data to an output file
- ▶ In Python, file handling consists of three steps:
  - ▶ Opening the file
  - ▶ Processing the file, i.e., performing read or write operations
  - ▶ Closing the file

# File Types

- ▶ There are two major types of files:
  - ▶ **Text files** – files whose contents can be viewed using a text editor
    - ▶ A text file is a sequence of ASCII or Unicode characters
    - ▶ Examples for text files include Python programs, HTML pages, and text documents
  - ▶ **Binary files** – files that store the data in the same way as it is stored in memory
    - ▶ You can't read a binary file using a text editor
    - ▶ Examples for binary files include mp3 files, image files and word documents
- ▶ Storing the number 1234 in a binary vs a text file:

|         | byte 0   | byte 1   | byte 2   | byte 3   |
|---------|----------|----------|----------|----------|
| Binary: | 00000000 | 00000000 | 00000100 | 11010010 |
|         | 0        | 0        | 4        | 210      |
| Text:   | 49       | 50       | 51       | 52       |

# Opening a File

- ▶ The function **open()** is used to open a file and create a file object
- ▶ Its syntax is: `fileobject = open(filename, mode)`
  - ▶ *filename* may be given as an absolute path, or as a path relative to the directory in which the program is being executed
  - ▶ *mode* is a string that specifies the type of the operation you want to perform on the file
- ▶ For example, to open a file for text-mode writing:

```
f = open("myfile.txt", "w")
```

- ▶ In Windows, remember to escape backslashes while using absolute path names:  
`f = open("C:\\\\Users\\\\roi\\\\Documents\\\\README.md", "w")`
- ▶ Or you can use a “raw string” by specifying the **r** character in front of the string:  
`f = open(r"C:\\\\Users\\\\roi\\\\Documents\\\\README.md", "w")`

# File Modes

- ▶ The following table lists the different modes available to you:

| mode argument | Description                                                             |
|---------------|-------------------------------------------------------------------------|
| r             | text, read-only (the default)                                           |
| w             | text, write (an existing file with the same name will be overwritten)   |
| a             | text, append to an existing file                                        |
| r+            | text, reading and writing                                               |
| rb            | binary, read-only                                                       |
| wb            | binary, write (an existing file with the same name will be overwritten) |
| ab            | binary, append to an existing file                                      |
| rb+           | binary, reading and writing                                             |

# Closing a File

- ▶ File objects are closed with the **close()** method as follows:

```
f.close()
```

- ▶ Closing a file releases valuable system resources
- ▶ Python closes any open file objects automatically when a program terminates or when the file object is no longer referenced in the program
- ▶ However, it is a good practice to close a file once you are done working with it

# Writing to a Text File

- ▶ The **write()** method of a file object writes a *string* to the file and returns the number of characters written:

```
f = open("myfile.txt", "w")
```

```
f.write("Hello world")
```

```
11
```

```
f.flush()
```

- ▶ The characters are written to an internal buffer before they are written to the file
- ▶ The method **flush()** flushes the internal buffer
  - ▶ Python automatically flushes the files when closing them
- ▶ You can check the default buffer size by calling:

```
import io
print (io.DEFAULT_BUFFER_SIZE)
```

```
8192
```

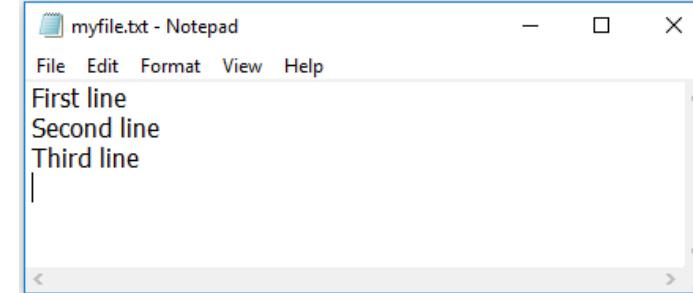
# Writing to a Text File

- ▶ Note that unlike print() function, the write() method doesn't print a newline character (\n) at the end of the string
- ▶ If you want to write multiple lines to the file, you should append \n to each line:

```
f = open("myfile.txt", "w")

f.write("First line\n")
f.write("Second line\n")
f.write("Third line\n")

f.close()
```



# Writing to a Text File

- More helpfully, the `print()` built-in takes an argument, *file*, to specify where to redirect its output:

```
f = open("myfile.txt", "w")

print("First line", file=f)
print("Second line", file=f)
print("Third line", file=f)

f.close()
```

- This program produces the same output as before, the only difference is that here the newline character (`\n`) is automatically added by the `print()` function.

# Reading from a Text File

- ▶ There are a few methods available for reading from a file:
  - ▶ **read(n)** – reads n bytes from the file
    - ▶ In ASCII text files, this means reading n characters from the file
    - ▶ If n is omitted, the entire file is read in
  - ▶ **readline()** – reads a single line the file, up to and including the newline character
  - ▶ **readlines()** – reads all of the lines into a list of strings in one go
- ▶ For example, to read all the file at once:

```
f = open("myfile.txt", "r")

print(f.read()) # read all content at once

f.close()
```

First line  
Second line  
Third line

# Reading from a Text File

- ▶ Example for reading the data in chunks:

```
f = open("myfile.txt", "r")

print("First chunk: ", f.read(4)) # read the first 4 characters
print("Second chunk: ", f.read(10)) # read the next 10 characters
print("Third chunk: ", f.read()) # read the remaining characters

f.close()
```

```
First chunk: Firs
Second chunk: t line
Sec
Third chunk: ond line
Third line
```

# Reading from a Text File

- ▶ Example for reading the file line by line using **readline()**:

```
f = open("myfile.txt", "r")

line = f.readline()
while line:
 print(line, end="")
 line = f.readline()

f.close()
```

First line  
Second line  
Third line

- ▶ Both `read()` and `readline()` return an empty string when they reach the end of the file
- ▶ Because `line` retains its newline character when read in, we use `end=""` to prevent `print` from adding another, which would be output as a blank line

# Reading from a Text File

- ▶ File objects are iterable, and looping over a (text) file returns its lines one at a time:

```
f = open("myfile.txt", "r")

for line in f:
 print(line, end="")

f.close()
```

First line  
Second line  
Third line

# Appending Data to a Text File

- ▶ We can use "a" mode to append data to end of the file
- ▶ The following program demonstrates how to append data to the end of the file:

```
f = open("myfile.txt", "a")

print("Fourth line", file=f)
print("Fifth line", file=f)

f.close()
```

```
f = open("myfile.txt", "r")

for line in f:
 print(line, end="")

f.close()
```

```
First line
Second line
Third line
Fourth line
Fifth line
```

## Exercise (19)

- ▶ Write a program that writes the first four powers of the numbers between 1 and 1,000 in comma-separated fields to the file powers.txt
- ▶ The file contents should look like:

```
1, 1, 1, 1
2, 4, 8, 16
3, 9, 27, 81
...
999, 998001, 997002999, 996005996001
1000, 1000000, 1000000000, 100000000000
```

- ▶ Then read the numbers from the file powers.txt and print a list of the cubes of all the numbers between 1 and 100

## Exercise (20)

---

- ▶ The coast redwood tree species, *Sequoia sempervirens*, includes some of the oldest and tallest living organisms on Earth
- ▶ Some details concerning individual trees are given in the tab-delimited text file redwood-data.txt at <https://goo.gl/KnFPfC>
- ▶ Write a Python program to read in this data and report the tallest tree and the tree with the greatest diameter

## Exercise (21)

- ▶ The novel *Moby-Dick* is out of copyright and can be downloaded as a text file from the Project Gutenberg website at <http://www.gutenberg.org/files/2701/>
- ▶ Write a program to output the 100 words most frequently used in the book by storing a count of each word encountered in a dictionary
- ▶ *Hint:* use Python's string methods to strip out any punctuation
  - ▶ It suffices to replace any of the following characters with the empty string: !?"';,().'\*[]
- ▶ Compare the frequencies of the top 100 words with the prediction of *Zipf's Law*:

$$\log f(w) = \log C - a \log r(w)$$

- ▶ where  $f(w)$  is the number of occurrences of word  $w$ ,  $r(w)$  is the corresponding rank (1 = most common, 2 = second most common, etc.) and  $C$  and  $a$  are constants
- ▶ Typically,  $C = \log f(w_1)$ , where  $w_1$  is the most common word, and  $a = 1$

# The with Statement

- ▶ The **with** statement creates a block of code that is executed within a certain *context*
- ▶ A context is defined by a *context manager* that provides a pair of methods (`__enter__` and `__exit__`) describing how to enter and leave the context
- ▶ For example, the file object returned by the `open()` method is a context manager
- ▶ It defines an exit method which is called on exiting the context, and simply closes the file, so this doesn't need to be done explicitly
- ▶ To open a file within a context, use:

```
with open(filename, mode) as f:
 # process the file in some way, for example:
 lines = f.readlines()
```

- ▶ The scope of the file object `f` is limited to the body of the `with` statement
- ▶ The context manager ensures that the file is closed, even if something goes wrong in the block

# The with Statement

- ▶ The following example shows how to read a file within a with statement:

```
with open("myfile.txt", "r") as f:
 for line in f:
 print(line, end="")
```

```
First line
Second line
Third line
Fourth line
Fifth line
```

# Working with Unicode Text Files

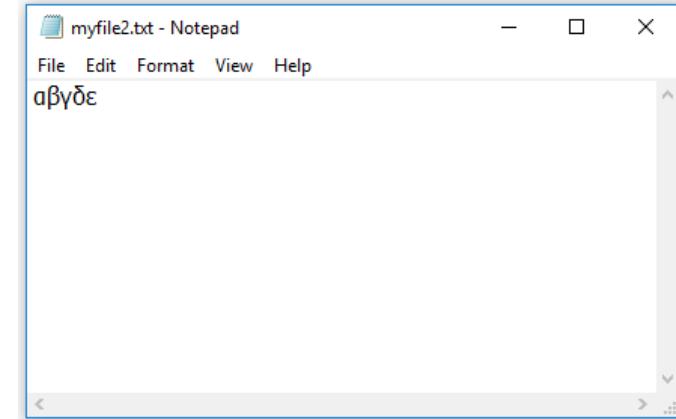
- To work with Unicode text files, you need to open the file using the `open()` function from the `io` module, and specify the *encoding* argument:

```
import io

with io.open("myfile2.txt", "w", encoding="utf8") as f:
 print("αβγδε", file=f)
```

```
with io.open("myfile2.txt", "r", encoding="utf8") as f:
 print(f.read())
```

αβγδε



# Working with Binary Files

- ▶ To write to a binary file, you need to open the file for writing in binary mode and then write data to the file as hexadecimal strings:

```
f = open("myfile.bin", "wb")

f.write(b"\x0a\x1b\x2c")
f.write(b"\x3d\x4e\x5f")

f.close()
```

- ▶ To convert Python values to hexadecimal strings, you need to use the **struct** module
- ▶ The struct module performs conversions between Python values and C structs represented as Python strings
  - ▶ This module is useful in handling binary data stored in files or from network connections among other sources

# Working with Binary Files

- ▶ **struct.pack(fmt, v1, v2, ...)** returns a string containing the values v1, v2, ... packed according to the given format
- ▶ **struct.unpack(fmt, string)** unpacks the string according to the given format
  - ▶ The result is a tuple even if it contains exactly one item
  - ▶ The string must contain exactly the amount of data required by the format

```
import struct

f = open("myfile.bin", "wb")
f.write(struct.pack('I', 23250))
f.write(struct.pack('?', True))
f.write(struct.pack('5s', "Hello".encode()))

f.close()
```

```
f = open("myfile.bin", "rb")

print(struct.unpack('I', f.read(4))[0])
print(struct.unpack('?', f.read(1))[0])
print(struct.unpack('5s', f.read(5))[0].decode("UTF-8"))

f.close()
```

```
23250
True
Hello
```

# Format Characters

- The format strings describe the layout of the C structs and the intended conversion to/from Python values:

| Format | C Type             | Python type        | Standard size |
|--------|--------------------|--------------------|---------------|
| x      | pad byte           | no value           |               |
| c      | char               | string of length 1 | 1             |
| b      | signed char        | integer            | 1             |
| B      | unsigned char      | integer            | 1             |
| ?      | _Bool              | bool               | 1             |
| h      | short              | integer            | 2             |
| H      | unsigned short     | integer            | 2             |
| i      | int                | integer            | 4             |
| I      | unsigned int       | integer            | 4             |
| l      | long               | integer            | 4             |
| L      | unsigned long      | integer            | 4             |
| q      | long long          | integer            | 8             |
| Q      | unsigned long long | integer            | 8             |
| f      | float              | float              | 4             |
| d      | double             | float              | 8             |
| s      | char[]             | string             |               |
| p      | char[]             | string             |               |
| P      | void *             | integer            |               |

# Working with Binary Files

- ▶ To write to a binary file, you need to open the file for writing in binary mode and then write data to the file as hexadecimal strings:

```
f = open("myfile.bin", "wb")

f.write(b"\x0a\x1b\x2c")
f.write(b"\x3d\x4e\x5f")

f.close()
```

- ▶ To convert Python values to hexadecimal strings, you need to use the **struct** module
- ▶ The struct module performs conversions between Python values and C structs represented as Python strings
  - ▶ This module is useful in handling binary data stored in files or from network connections among other sources

## Exercise (22)

- ▶ Define a function **copy\_file(source, dest)** that takes the paths of a source file and a destination file, and copies the binary data from the source file to the destination file
- ▶ Note: the source file could be large, and might not fit entirely in memory
- ▶ Usage example:

```
copy_file("source.jpg", "dest.jpg")
```

```
2048 bytes copied successfully
700 bytes copied successfully
```

# Errors and Exceptions

- ▶ Python distinguishes between two types of error:
- ▶ **Syntax errors** are mistakes in the grammar of the language and are checked before the program is executed
  - ▶ Syntax errors are always fatal: there is nothing the Python compiler can do for you if your program does not conform to the grammar of the language
- ▶ **Exceptions** are *runtime errors*: conditions usually caused by attempting an invalid operation on an item of data (such as division by zero)
  - ▶ Mechanisms exist for “catching” runtime errors and the condition gracefully without stopping the program’s execution

# Syntax Errors

- ▶ Syntax errors are caught by the Python compiler and produce a message indicating where the error occurred
- ▶ For example:

```
for lambda in range(8):

 File "<ipython-input-1-f6fc4897d4ad>", line 1
 for lambda in range(8):
 ^
SyntaxError: invalid syntax
```

- ▶ Because `lambda` is a reserved keyword, it cannot be used as a variable name. Its occurrence where a variable name is expected is therefore a syntax error.

# Syntax Errors

- ▶ Because a line of Python code may be split within an open bracket ("()", "[]", or "{}"), a statement split over several lines can sometimes cause a SyntaxError to be indicated somewhere other than the location of the true bug:

```
a = [1, 2, 3, 4,
b = 5

File "<ipython-input-2-2ff941d04ea6>", line 2
 b = 5
 ^
SyntaxError: invalid syntax
```

- ▶ Here, the statement `b = 5` is syntactically valid: the error arises from failing to close the square bracket of the previous list declaration
- ▶ There are two special types of SyntaxError that are worth mentioning:
  - ▶ **IndentationError** occurs when a block of code is improperly indented
  - ▶ **TabError** is raised when tabs and spaces are mixed inconsistently to provide indentation

# Exceptions

- ▶ An exception occurs when an syntactically correct expression is executed and causes a **runtime error**
- ▶ There are different types of built-in exceptions, and custom exceptions can be defined by the programmer if required
- ▶ If an exception is not “caught” using the try ... except clause described later, it halts the execution of the program
- ▶ When a runtime error occurs, Python creates an exception object which contains all the relevant information about the error just occurred
- ▶ This object contains an error message and a **stack traceback**: the history of function calls leading to the error is reported so that its location in the program execution can be determined

# Common Python Exceptions

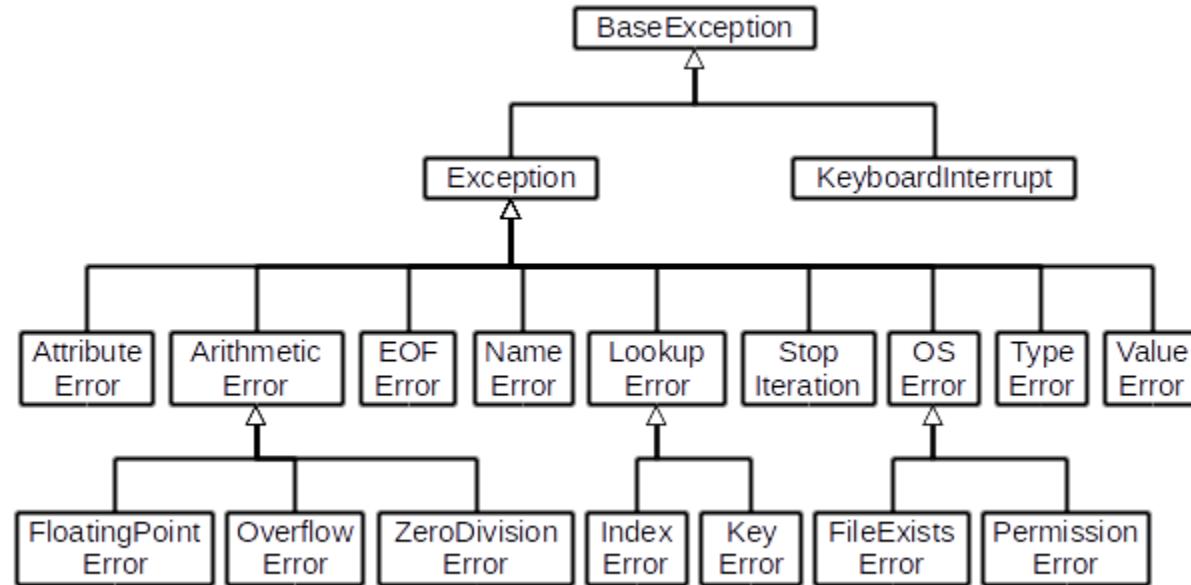
- ▶ A list of the more commonly encountered built-in exceptions and their descriptions:

| Exception             | Cause and Description                                                                                                             |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| FileNotFoundException | Attempting to open a file or directory that does not exist, inherits from OSError                                                 |
| IndexError            | Indexing a sequence (such as a list or string) with a subscript that is out of range                                              |
| KeyError              | Indexing a dictionary with a key that does not exist in that dictionary                                                           |
| NameError             | Referencing a local or global variable name that has not been defined                                                             |
| TypeError             | Attempting to use an object of an inappropriate type as an argument to a built-in operation or function                           |
| ValueError            | Attempting to use an object of the correct type but with an incompatible value as an argument to a built-in operation or function |
| ZeroDivisionError     | Attempting to divide by zero                                                                                                      |
| SystemExit            | Raised by the sys.exit function – if not handled, this function causes the Python interpreter to exit                             |

- ▶ The full list can be found at <https://docs.python.org/3/library/exceptions.html>

# Exception Class Hierarchy

- ▶ The **BaseException** class is the root of all exception classes in Python
- ▶ The following figure shows exception class hierarchy in Python:



# Common Python Exceptions

- ▶ A **NameError** exception occurs when a variable name is used that hasn't been:

```
print("x = ", x)

NameError Traceback (most recent call last)
<ipython-input-1-189b4fb9adac> in <module>()
----> 1 print("x = ", x)

NameError: name 'x' is not defined
```

- ▶ A division by zero causes a **ZeroDivisionError**:

```
a, b = 5, 0
a / b

ZeroDivisionError Traceback (most recent call last)
<ipython-input-2-a827eb207e9c> in <module>()
 1 a, b = 5, 0
----> 2 a / b

ZeroDivisionError: division by zero
```

# Common Python Exceptions

- ▶ Trying to access an element at invalid index causes an **IndexError**:

```
list1 = [11, 3, 99, 15]
list1[5]

IndexError Traceback (most recent call last)
<ipython-input-3-16d803b33219> in <module>()
 1 list1 = [11, 3, 99, 15]
----> 2 list1[5]

IndexError: list index out of range
```

- ▶ Trying to open a file that doesn't exist in read mode causes a **FileNotFoundException**:

```
f = open("filedoesnotexist.txt", "r")

FileNotFoundException Traceback (most recent call last)
<ipython-input-4-00ca043a59e9> in <module>()
----> 1 f = open("filedoesnotexist.txt", "r")

FileNotFoundException: [Errno 2] No such file or directory: 'filedoesnotexist.txt'
```

# Common Python Exceptions

- ▶ A **TypeError** is raised if an object of the wrong type is used in an expression or function, for example when trying to add a string to an integer:

```
10 + "12"

TypeError Traceback (most recent call last)
<ipython-input-6-db2f06a66b01> in <module>()
----> 1 10 + "12"

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

- ▶ A **ValueError**, on the other hand, occurs when the object involved has the correct *type* but an invalid *value*:

```
float("hello")

ValueError Traceback (most recent call last)
<ipython-input-7-7124e8e12e61> in <module>()
----> 1 float("hello")

ValueError: could not convert string to float: 'hello'
```

# Traceback Report

- When an exception is raised but not handled, Python will issue a traceback report indicating where in the program flow it occurred:

```
def lumberjack():
 bright_side_of_death()

def bright_side_of_death():
 return tuple()[0]

lumberjack()

IndexError Traceback (most recent call last)
<ipython-input-10-81211c2c9fdb> in <module>()
 5 return tuple()[0]
 6
----> 7 lumberjack()

<ipython-input-10-81211c2c9fdb> in lumberjack()
 1 def lumberjack():
----> 2 bright_side_of_death()
 3
 4 def bright_side_of_death():
 5 return tuple()[0]

<ipython-input-10-81211c2c9fdb> in bright_side_of_death()
 3
 4 def bright_side_of_death():
----> 5 return tuple()[0]
 6
 7 lumberjack()

IndexError: tuple index out of range
```

# Handling Exceptions

- ▶ To catch an exception in a block of code, write the code within a **try:** clause and handle any exceptions raised in an **except:** clause, using the following syntax:

```
try:
 # try block
 # write code that might raise an exception here
 <statements>
except ExceptionType:
 # except block
 # handle exception here
 <exception handler>
```

- ▶ When an exception occurs in the try block, execution of the rest of the try block is skipped
  - ▶ If the exception raised matches the exception type in the except clause, the corresponding handler is executed
  - ▶ If the exception doesn't match the exception type in the except clause, the program halts with a traceback report
- ▶ On the other hand, If no exception is raised in the try block, the except clause is skipped

# Handling Exceptions

- ▶ The following example catches a ZeroDivisionError exception:

```
try:
 num = int(input("Enter a number: "))
 result = 10 / num
 print("Result: ", result)
except ZeroDivisionError:
 print("Can't divide a number by zero")
```

```
Enter a number: 0
Can't divide a number by zero
```

- ▶ If we run the program again and enter a string instead of a number, then a ValueError exception is raised, which is *unhandled* by our except clause:

```
Enter a number: hello

ValueError Traceback (most recent call last)
<ipython-input-3-834c05ed2c52> in <module>()
 1 try:
----> 2 num = int(input("Enter a number: "))
 3 result = 10 / num
 4 print("Result: ", result)
 5

ValueError: invalid literal for int() with base 10: 'hello'
```

# Handling Multiple Exceptions

- ▶ We can add as many except clauses as we want to handle different types of exceptions. The general format of such a try-except statement is as follows:

```
try:
 # write code that might raise an exception here
except <ExceptionType1>:
 # handle ExceptionType1 here
except <ExceptionType2>:
 # handle ExceptionType2 here
...
except:
 # handle any other type of exception here
```

- ▶ When exception occurs, Python matches the exception raised against every except clause sequentially
- ▶ If a match is found then the handler in the corresponding except clause is executed and rest of the of the except clauses are skipped
- ▶ In case the exception raised doesn't match any except before the last except clause, then the handler in the last except clause is executed

# Handling Multiple Exceptions

- ▶ Example for a try block with multiple except blocks:

```
try:
 num = int(input("Enter a number: "))
 result = 10 / num
 print("Result: ", result)
except ZeroDivisionError:
 print("Can't divide a number by zero")
except ValueError:
 print("Only integers are allowed")
except:
 print("Some unexpected error occurred")
```

```
Enter a number: hello
Only integers are allowed
```

# Handling Multiple Exceptions

- ▶ Another example for handling exceptions when reading data from a file:

```
filename = input("Enter file name: ")

try:
 f = open(filename, "r")
 for line in f:
 print(line, end="")
 f.close()
except FileNotFoundError:
 print("File not found")
except PermissionError:
 print("You don't have permission to read the file")
except:
 print("Unexpected error while reading the file")
```

```
Enter file name: test
File not found
```

# Handling Multiple Exceptions

- To handle more than one exception in a single except block, list them in a tuple (which must be within brackets)

```
try:
 num = int(input("Enter a number: "))
 result = 10 / num
 print("Result: ", result)

except (ZeroDivisionError, ValueError):
 print("The input is zero or not numeric!")
```

```
Enter a number: hello
The input is zero or not numeric!
```

# The else and finally clauses

- ▶ The try ... except statement has two more optional clauses (which must follow any except clauses):
  - ▶ Statements in a block following the **else** keyword are executed if an exception was not raised
  - ▶ Statements in a block following the **finally** keyword are always executed, whether an exception was raised or not
- ▶ The use of the **else** clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the try ... except statement
- ▶ The finally block is used to make sure files or resources are closed or released regardless of whether an exception occurs, even if you don't catch the exception

# The else and finally clauses

- ▶ The following code gives an example of a try ... except ... else ... finally clause:

```
def process_file(filename):
 try:
 f = None
 f = open(filename, "r")
 except IOError:
 print("Couldn't open {} for reading".format(filename))
 return
 else:
 # we don't want to catch IOError here if it's raised
 for line in f:
 print(line, end="")
 finally:
 # close the file if it was successfully opened
 if f:
 f.close()
```

```
process_file("test.txt")
```

```
Couldn't open test.txt for reading
```

```
process_file("myfile.txt")
```

```
First line
Second line
Third line
```

# Ask Forgiveness, Not Permission

- ▶ EAFP (easier to ask for forgiveness than permission) is a common Python coding style that assumes the existence of files, attributes, valid keys, etc., and catches exceptions if the assumption proves false
- ▶ This clean style is characterized by the presence of many try and except statements:

## Ask for permission

```
if can_do_operation():
 perform_operation()
else:
 handle_error_case()
```

## Ask for forgiveness

```
try:
 perform_operation()
except UnableToPerform:
 handle_error_case()
```

- ▶ E.g., asking for forgiveness when accessing dictionary keys that may not exist:

```
contacts = {"Tom": "122-444-333", "Jim": "412-1231-121", "Ron": "891-121-1212"}

name = input("Enter a name: ")
try:
 print(contacts[name])
except KeyError:
 print("Sorry, there is no contact with the name {}".format(name))
```

```
Enter a name: Joe
Sorry, there is no contact with the name Joe
```

# Don't Do

- ▶ You may come across the following type of construction:

```
try:
 [do something]
except: # Don't do this!
 pass
```

- ▶ This will execute the statements in the try block and ignore *any* exceptions raised – it is very unwise to do this as it makes code very hard to maintain and debug
- ▶ Always catch specific exceptions and handle them appropriately, allowing any other exceptions to “bubble up” to be handled (or not) by any other except clauses

## Exercise (25)

- ▶ Write a program to read in the data from the file swallow-speeds.txt at <https://goo.gl/KnFPfC> and use it to calculate the average air-speed velocity of an (unladen) African swallow
- ▶ Use exceptions to handle the processing of lines that do not contain valid data points

# Raising Exceptions

- ▶ Sometimes it is desirable for a program to raise a particular exception if some condition is met
- ▶ A function raises exception by creating an exception object from the appropriate class and using the **raise** keyword to throw the exception to the calling code:

```
raise SomeExceptionClass("Error message describing cause of the error")
```

- ▶ When an exception is raised inside a function and is not caught there, it is automatically propagated to the calling function (and any function up in the stack), until it is caught by a try-except statement in some calling function
- ▶ If the exception reaches the main module and still not handled, the program terminates with an error message

# Raising Exceptions

- ▶ For example, let's define a function to calculate the factorial of a number
- ▶ Factorial is only valid for positive integers, thus we can prevent passing data of any other type by checking the argument and raising the appropriate exception:

```
def factorial(n):
 if type(n) is not int:
 raise TypeError("Argument must be int")

 if n < 0:
 raise ValueError("Argument must be non-negative")

 f = 1
 for i in range(2, n + 1):
 f *= i
 return f
```

```
try:
 print("Factorial of 4 is:", factorial(4))
 print("Factorial of 1.5 is:", factorial(1.5))
except Exception:
 print("Invalid input")
```

```
Factorial of 4 is: 24
Invalid input
```

# Accessing the Exception Object

- ▶ We can access the exception object using the following form of except clause:

```
except ExceptionType as e
```

- ▶ The exception object will be assigned to the variable e
- ▶ For example:

```
try:
 print("Factorial of 4 is:", factorial(4))
 print("Factorial of 1.5 is:", factorial(1.5))
except Exception as e:
 print("Error:", e)
```

```
Factorial of 4 is: 24
Error: Argument must be int
```

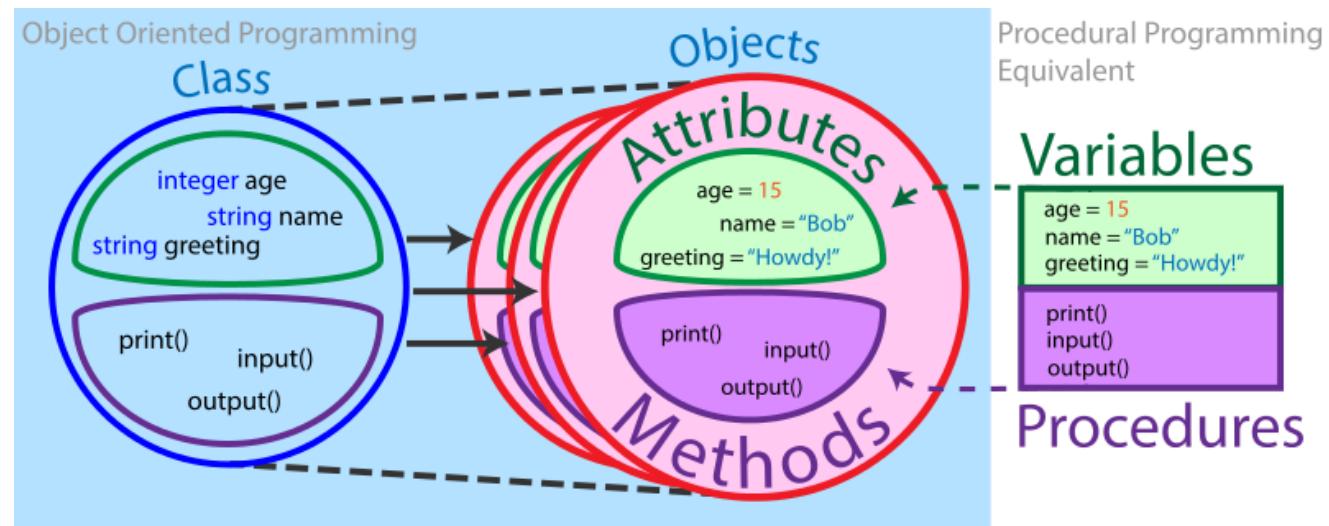
- ▶ Notice that the error message printed by the exception object is the same message that we passed while raising the exception

## Exercise (26)

- ▶ Python follows the convention of many computer languages in choosing to define  $0^0 = 1$
- ▶ Write a function, **power(a, b)**, which behaves the same as the Python expression  $a^{**}b$ , but raises a ValueError if a and b are both zero
- ▶ Also if a or b are not numbers, the function should raise a TypeError
- ▶ Write a proper docstring for the function including the description of the raised exceptions

# Object-Oriented Programming (OOP)

- ▶ OOP is a programming paradigm based on the concept of "objects"
- ▶ An object represents an entity in the program, which holds data about itself (**attributes**), and defines functions (**methods**) for manipulating the data
- ▶ An object is created (*instantiated*) from a “blueprint” called a **class**, which dictates its behavior by defining its attributes and methods



# Objects in Python

- ▶ In fact, as we have already pointed out, everything in Python is an object
- ▶ So, for example, a Python string is an instance of the `str` class
- ▶ A `str` object possesses its own data (the sequence of characters making up the string) and provides (“exposes”) a number of methods for manipulating that data
- ▶ For example, the `capitalize()` method returns a new string object created from the original string by capitalizing its first letter:

```
str = "hello world"
str.capitalize()
```

```
'Hello world'
```

- ▶ Even indexing a string is really to call the method `__getitem__`:

```
str.__getitem__(6)
```

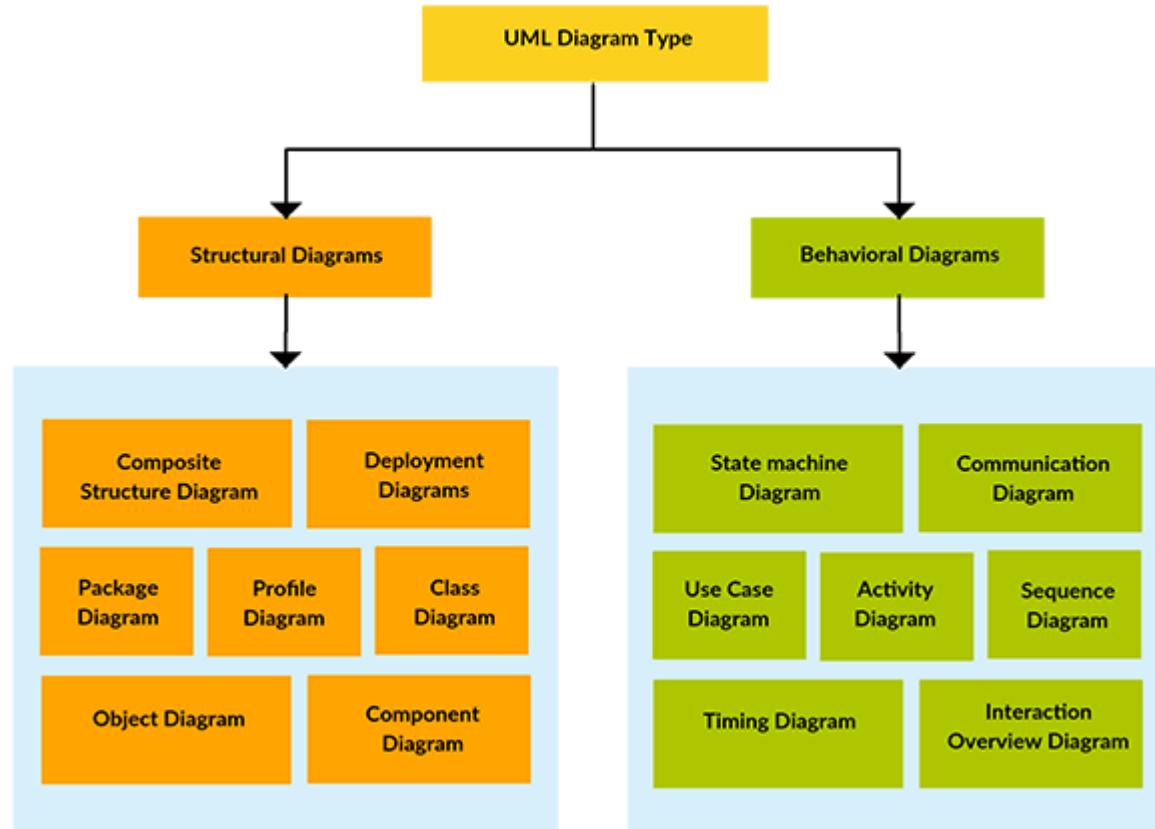
```
'w'
```

# Object-Oriented Design

- ▶ **Object-oriented design** is the process of planning a system of interacting objects for the purpose of solving a software problem
- ▶ Using the OOP approach you can break a problem down into units of data and operations that it is appropriate to carry out on that data
- ▶ For example, a retail bank deals with people who have bank accounts
- ▶ A natural object-oriented approach to managing a bank would be to define:
  - ▶ BankAccount class, with attributes such as an account number, balance and owner, and methods for allowing (or forbidding) transactions depending on its balance
  - ▶ Customer class with attributes such as a name, address, and date of birth, and methods for calculating the customer's age from their date of birth for example

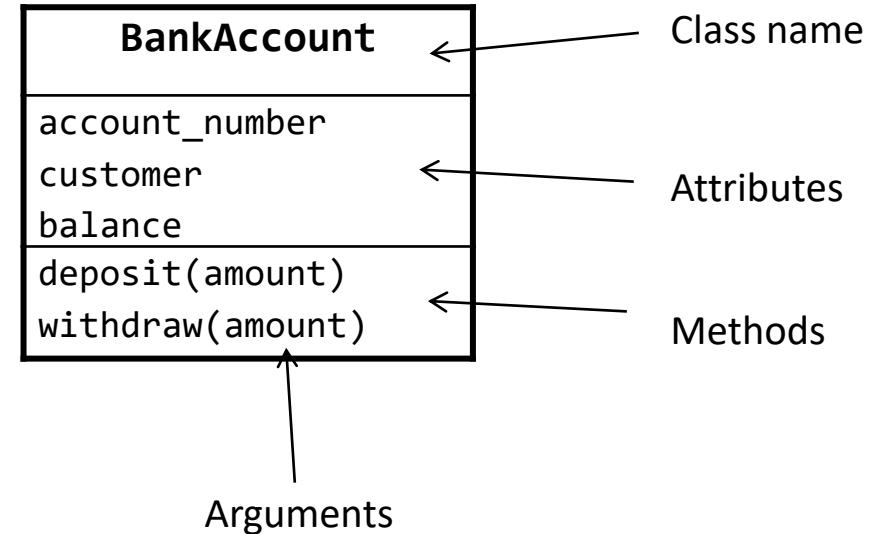
# UML

- ▶ **UML (Unified Modeling Language)** is a general-purpose, modeling language that provides a standard way to visualize the design of a system



# UML Class Diagram

- ▶ A **class diagram** describes the structure of a system by showing the system's classes, their attributes, methods, and the relationships among objects



# Object-Oriented Concepts

- ▶ **Information hiding/Encapsulation:** The ability to protect some components of the object from external entities
- ▶ **Inheritance:** The ability for a class to extend or override functionality of another class
- ▶ **Protocol/Interface:** A common means for unrelated objects to communicate with each other. These are definitions of methods and values which the objects agree upon in order to co-operate, part of an API.
- ▶ **Polymorphism:** The ability to replace an object with its subobjects
- ▶ **Dependency injection:** If an object depends upon having an instance of some other object, then the needed object is "injected" into the dependent object

# Defining a Class

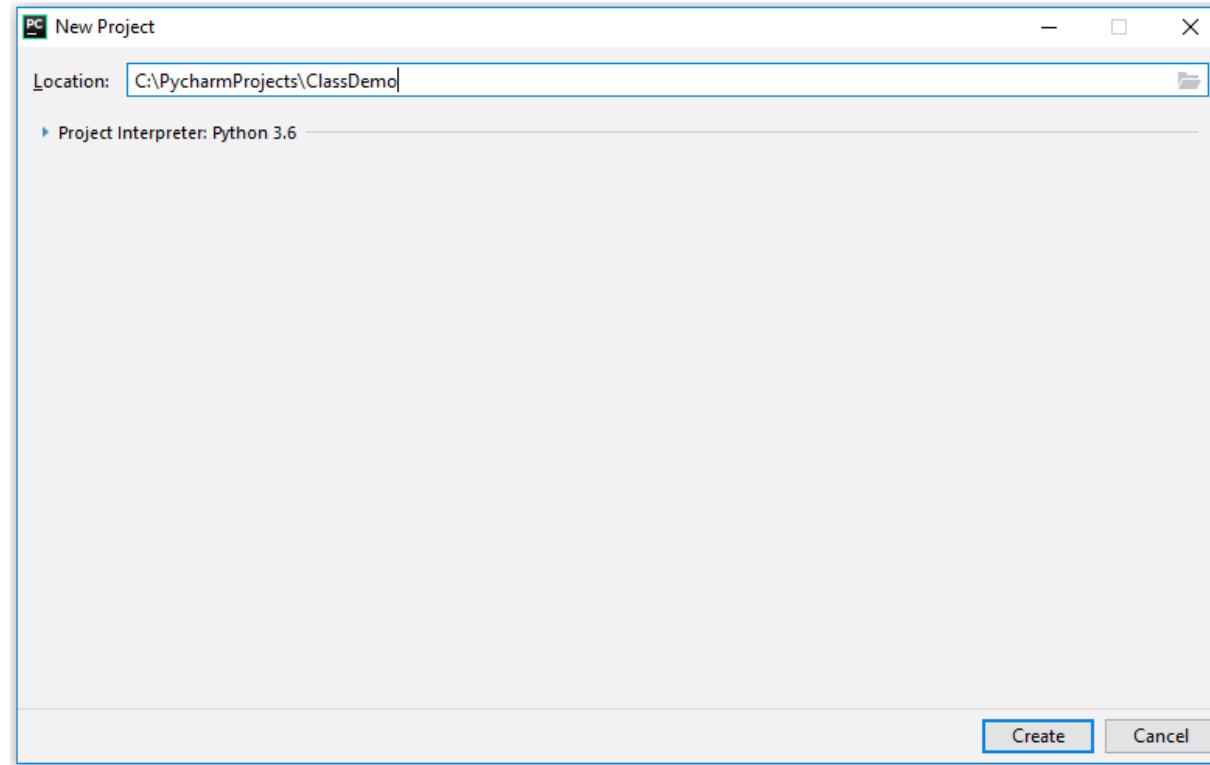
- ▶ A class is defined using the **class** keyword and indenting the body of statements (attributes and methods) in a block following this declaration:

```
class ClassName:
 <attribute_1 definition>
 ...
 <attribute_N definition>
 <method_1 definition>
 ...
 <method_N definition>
```

- ▶ It is conventional to write classes names in *CamelCase*
- ▶ It is a good idea to follow the **class** statement with a docstring describing what the class does
- ▶ Class methods are defined using the familiar **def** keyword, but the first argument to each method should be a variable named **self**, which refers to the object that invoked the method

# Class Example

- ▶ Create a new project in PyCharm named ClassDemo



- ▶ Add a file named bank\_account.py to the project

# Class Example

## ▶ Define the following BankAccount class:

```

class BankAccount:
 """A class representing a bank account"""
 def __init__(self, customer, account_number, balance=0):
 self.customer = customer
 self.account_number = account_number
 self.balance = balance

 def deposit(self, amount):
 """Deposit amount into the bank account"""
 if amount > 0:
 self.balance += amount
 else:
 print("Invalid deposit amount:", amount)

 def withdraw(self, amount):
 """Withdraw amount from the bank account, ensuring there are
 sufficient funds """
 if amount > 0:
 if amount > self.balance:
 print("Insufficient funds")
 else:
 self.balance -= amount
 else:
 print("Invalid withdrawal amount:", amount)

```

- `__init__()` is special method called **initializer** or **constructor**
- It is invoked every time after a new object is created in memory
- Its purpose is to create the object's attributes with some initial values

Inside the class we use **self** to access the object's attributes and methods

# Importing the Class

- ▶ To use the BankAccount class in a Notebook, copy the script bank\_account.py to C:\Notebooks
- ▶ Create a new Notebook named TestBankAccount
- ▶ Import the BankAccount class from bank\_account.py by writing:

```
from bank_account import BankAccount
```
- ▶ This notebook can now create BankAccount objects and manipulate them by calling the methods described earlier

# Creating Objects

- ▶ An *instance* of a class is created with the syntax:

```
object = className(args)
```

- ▶ If the class has an `__init__` method, the arguments must match the parameters of the `__init__` method (without `self`)
- ▶ In our example, an account is opened by creating a `BankAccount` object:

```
my_account = BankAccount("Joe Smith", 34572881)
```

- ▶ This statement does the following:
  - ▶ Creates an object of the `BankAccount` class
  - ▶ Invokes the `__init__` method, passing the newly created `BankAccount` object to `self`, and the values "Joe Smith" and 3452881 to the `customer` and `account_id` parameters of `__init__`
  - ▶ Adds the attributes `customer`, `account_number`, and `balance` to the newly created object (`balance` gets the default value 0)
  - ▶ Assigns the reference of the `BankAccount` object to the variable `my_account`

# Attributes and Methods

- Both attributes and methods of the object are accessed using the dot notation:

```
object.attribute # access an attribute
object.method(arguments) # call a method
```

- For example:

```
my_account.account_number # access an attribute of my_account
```

```
34572881
```

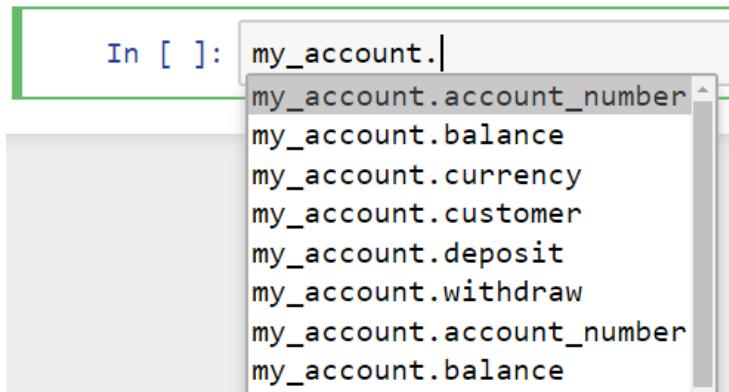
```
my_account.deposit(250) # call a method of my_account
```

```
my_account.balance
```

```
250
```

# Attributes and Methods

- ▶ In Jupyter Notebook, after typing the object name and dot (.), you can press Tab to examine all the object's attributes and methods:



```
In []: my_account.|
```

The screenshot shows a Jupyter Notebook cell with the input "In [ ]: my\_account.|". A dropdown menu is open, listing the attributes and methods of the "my\_account" object. The listed items are: my\_account.account\_number, my\_account.balance, my\_account.currency, my\_account.customer, my\_account.deposit, my\_account.withdraw, my\_account.account\_number, and my\_account.balance. The first item, "my\_account.account\_number", is highlighted.

# Attributes and Methods

- ▶ We can create as many objects as we want, each object with its own set of attributes
- ▶ Changing the attributes for one object will not affect the attributes of other objects

```

account1 = BankAccount("Customer A", 100)
account2 = BankAccount("Customer B", 200)
account3 = BankAccount("Customer C", 300)

print("The addresses of the account objects")
print("account1:", id(account1))
print("account2:", id(account2))
print("account3:", id(account3))
print()

print("The addresses of the account number attributes")
print("account1.account_number:", id(account1.account_number))
print("account2.account_number:", id(account2.account_number))
print("account3.account_number:", id(account3.account_number))

```

The addresses of the account objects

```

account1: 2490814268192
account2: 2490814268136
account3: 2490814268248

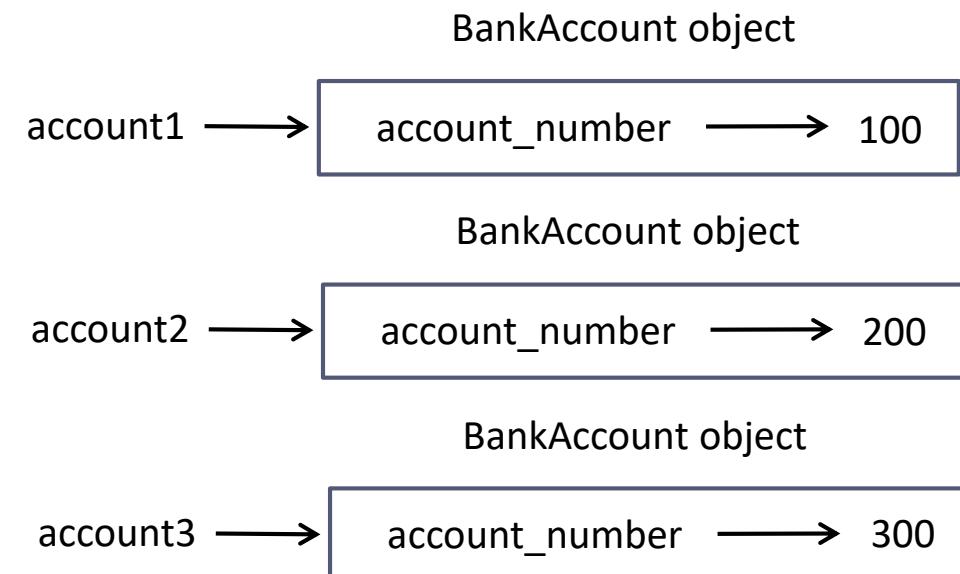
```

The addresses of the account number attributes

```

account1.account_number: 2000190016
account2.account_number: 2000193216
account3.account_number: 2490814178000

```



# Passing Objects as Arguments to Functions

- ▶ Just like built-in objects, we can pass objects of user-defined classes to functions
- ▶ The following shows how to pass BankAccount objects to a function named `print_account_info()`

```
def print_account_info(account):
 print("#####")
 print("Customer:", account.customer)
 print("Account number:", account.account_number)
 print("Balance:", account.balance)
 print("#####")
```

```
print_account_info(my_account)
```

```
#####
Customer: Joe Smith
Account number: 34572881
Balance: 250
#####
```

## Exercise

- ▶ Champion Motors Ltd. both sells and rents vehicles
- ▶ It rents two types of vehicles – Car and Bus
- ▶ Every vehicle has a license number, a make, a model and price
- ▶ Write the class Vehicle that represents a motor vehicle, and includes:
  - ▶ A constructor that gets (as parameters) a vehicle's number, a make, a model and price, and initializes the attributes of the vehicle
  - ▶ An method named `get_vehicle_name()` that returns the vehicle's name in the form make, model (e.g. “Suzuki Swift”)
  - ▶ A `display()` method which prints the vehicle's details in the following format:

```
License #: DL9087
Vehicle Name: Suzuki Swift
Price: $15,990.00
```

## Exercise

- ▶ Create a Notebook that defines a list with the following vehicles, and display the list contents:

| License # | Vehicle Name    | Price       |
|-----------|-----------------|-------------|
| FG1000    | Honda Civic     | \$21,875    |
| GH7000    | Mercedes BClass | \$42,700    |
| AY3000    | Bugatti Veyron  | \$1,500,000 |
| HI2000    | Hyundai Elantra | \$16,950    |

# Class and Instance Variables

- ▶ **Instance variable** is a variable for which each object of the class has a separate copy
  - ▶ e.g., customer, account\_id and balance in the BankAccount class are instance variables
- ▶ **Class variable** is shared by all instances of the class
  - ▶ They are defined inside the class but outside any of its methods
- ▶ Both types of variables are accessed using the object.attr notation
- ▶ For example, let's add to our BankAccount class a class variable named currency, and a third method, for printing the balance of the account

# Class and Instance Variables

```
class BankAccount:
 """A class representing a bank account"""
 currency = '$' # a class variable

 def __init__(self, customer, account_number, balance=0):
 ...

 def deposit(self, amount):
 ...

 def withdraw(self, amount):
 ...

 def check_balance(self):
 """Print a statement of the account balance."""
 print(f"The balance of account number {self.account_number} is "
 f"{self.currency}{self.balance:.2f}")
```

```
my_account = BankAccount("Joe Smith", 34572881)
my_account.deposit(525)
```

```
my_account.check_balance()
```

The balance of account number 34572881 is \$525.00

# Static Methods

- ▶ A **static method** belongs to the class rather than an object of the class
- ▶ A static method can be invoked without the need for creating an instance of a class
- ▶ To declare a static method, use the following syntax:

```
class C:
 @staticmethod
 def f(arg1, arg2, ...): ...
```

- ▶ `@staticmethod` is an example for a function decorator
  - ▶ Decorators are functions which modify the functionality of another function
- ▶ A static method doesn't receive an **implicit self argument**
  - ▶ Thus it cannot access instance variables
- ▶ A static method can be called either on the class (`C.f()`) or on an instance (`C().f()`)
- ▶ Use static methods sparingly
  - ▶ In many cases it's clearer to define module-level functions than static methods

# Static Method Example

- As an example for a static method, let's add the method `transfer_money()` to our `BankAccount` class that will transfer money between two bank accounts:

```
@staticmethod
def transfer_money(account1, account2, amount):
 """Transfer amount from account1 to account2"""
 if account1.withdraw(amount):
 account2.deposit(amount)
```

- We need to modify the `withdraw()` method to indicate whether it succeeded or not:

```
def withdraw(self, amount):
 """Withdraw amount from the bank account, ensuring there are sufficient funds."""
 if amount > 0:
 if amount > self.balance:
 print("Insufficient funds")
 else:
 self.balance -= amount
 return True
 else:
 print("Invalid withdrawal amount:", amount)
```

# Static Method Example

- ▶ Example for usage of transfer\_money():

```
from bank_account import BankAccount
```

```
account1 = BankAccount("Joe Smith", 34572881, 1000)
account2 = BankAccount("Helen Smith", 5799236)
```

```
BankAccount.transfer_money(account1, account2, 200)
```

```
account1.check_balance()
```

The balance of account number 34572881 is \$800.00

```
account2.check_balance()
```

The balance of account number 5799236 is \$200.00

```
BankAccount.transfer_money(account1, account2, 1000)
```

Insufficient funds

```
account1.check_balance()
```

The balance of account number 34572881 is \$800.00

# Class Methods

- ▶ Just like a static method, a class method is bound to the class rather than its object
- ▶ However, a class method receives the class as implicit first argument, just like an instance method receives the instance
- ▶ To declare a class method, use the following syntax:

```
class C:
 @classmethod
 def f(cls, arg1, arg2, ...): ...
```

- ▶ A class method can be called either on the class (`C.f()`) or on an instance (`C().f()`)
- ▶ The instance is ignored except for its class
- ▶ If a class method is called for a derived class, the derived class object is passed as the implied first argument

# Class Methods

- ▶ Class methods are typically used as factory methods, which generate objects for different use cases (like constructors)
- ▶ This allows you to provide alternate constructors for your class

```
class MyData:
 def __init__(self, data):
 """Initialize MyData from a sequence"""
 self.data = data

 @classmethod
 def from_file(cls, filename):
 """Initialize MyData from a file"""
 data = open(filename).readlines()
 return cls(data)

 @classmethod
 def from_dict(cls, data_dict):
 """Initialize MyData from a dict's items"""
 return cls(list(data_dict.items()))
```

```
from mydata import MyData
```

```
MyData([1, 2, 3]).data
```

```
[1, 2, 3]
```

```
MyData.from_file("test.txt").data
```

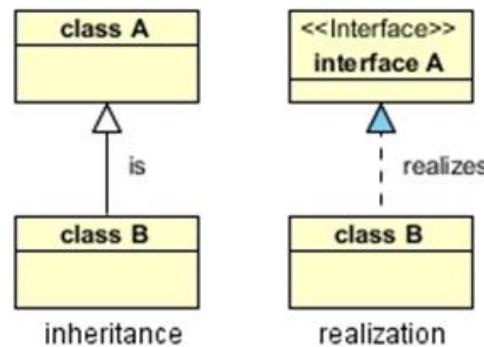
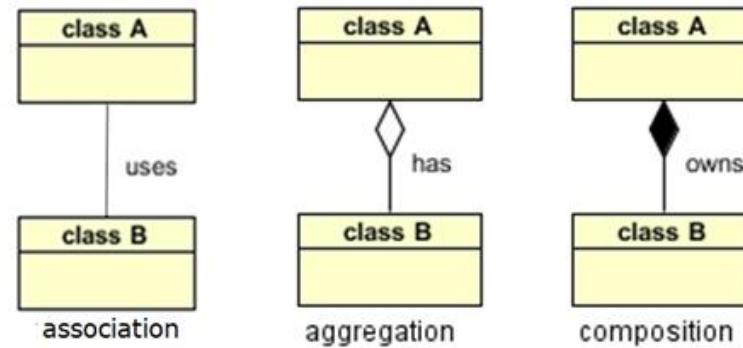
```
['First item\n', 'Second item\n', 'Third item']
```

```
MyData.from_dict({"spam": "ham"}).data
```

```
[('spam', 'ham')]
```

# Class Relationships

- ▶ There are five key relationships between classes:



# Class Relationships in Python

- ▶ **Composition** is represented when class A contains an object of class B and is responsible for its lifetime
- ▶ **Aggregation** is represented when class A stores a reference to class B for later use
- ▶ **Association** is represented when a reference to class B is passed as a method parameter to class A

```
class B: pass
class A:
 def __init__(self):
 self.b = B()
```

```
class B: pass
class A:
 def __init__(self, b):
 self.b = b

b = B()
a = A(b)
```

```
class B: pass
class A:
 def doSomething(self, b):
 pass

b = B()
a = A()
a.doSomething(b)
```

# Composition Example

- ▶ As an example for composition relationship, we'll define a class Rectangle that is composed of two point objects, which represent its top-left and bottom-right corners
- ▶ First, let's define a Point class with attributes x and y:

```
point.py
class Point:
 def __init__(self, x = 0, y = 0):
 self.x = x
 self.y = y
```

# Composition Example

- ▶ Now we'll define a Rectangle class, composed of two point objects:

```
rectangle.py
from point import Point

class Rectangle:
 def __init__(self, p1, p2):
 self.p1 = Point(p1.x, p1.y) # top-left corner
 self.p2 = Point(p2.x, p2.y) # bottom-right corner

 def get_width(self):
 return self.p2.x - self.p1.x

 def get_height(self):
 return self.p1.y - self.p2.y

 def get_area(self):
 return self.get_width() * self.get_height()
```

The rectangle object holds its own copy of the point objects

# Composition Example

- ▶ Creating a rectangle object and testing its methods:

```
from point import Point
from rectangle import Rectangle
```

```
p1 = Point(3, 10)
p2 = Point(7, 2)
r1 = Rectangle(p1, p2)
```

```
r1.get_width()
```

```
4
```

```
r1.get_height()
```

```
8
```

```
r1.get_area()
```

```
32
```

```
r1 is not affected from changing p1's coordinates
p1.x = 5
r1.get_width()
```

```
4
```

# Aggregation Example

---

- ▶ As an example for aggregation relationship, we'll define a Customer class with attributes name, address, and date of birth, and a method to calculate his/her age
- ▶ The BankAccount class will keep a reference to an instance of the Customer class, but will not controls the object's lifetime
- ▶ Note that it was possible to instantiate a BankAccount object by passing a string literal as customer
  - ▶ This is a consequence of Python's dynamic typing: no check is made that the object passed as an argument to the class constructor is of any particular type

# Aggregation Example

- ▶ First, let's define the Customer class:

```
customer.py
from datetime import datetime
class Customer:
 """A class representing a bank customer."""
 def __init__(self, name, address, date_of_birth):
 self.name = name
 self.address = address
 self.date_of_birth = datetime.strptime(date_of_birth, '%Y-%m-%d')

 def get_age(self):
 """Calculate and return the customer's age"""
 today = datetime.today()
 try:
 birthday = self.date_of_birth.replace(year=today.year)
 except ValueError:
 # birthday is 29 Feb but today's year is not a leap year
 birthday = self.date_of_birth.replace(year=today.year,
 day=self.date_of_birth.day - 1)
 if birthday > today:
 return today.year - self.date_of_birth.year - 1
 return today.year - self.date_of_birth.year
```

# Aggregation Example

- ▶ Then we can pass Customer objects to our BankAccount constructor:

```
from bank_account import BankAccount
from customer import Customer
```

```
customer1 = Customer("Helen Smith",
 "76 The Warren, Blandings, Sussex",
 "1976-02-29")
```

```
account1 = BankAccount(customer1, 2145288, 1000)
```

```
account1.customer.get_age()
```

42

```
print(account1.customer.address)
```

76 The Warren, Blandings, Sussex

# Class DocStrings

---

- ▶ The docstring for a class should summarize its behavior and list the public methods and instance variables
- ▶ The class constructor should be documented in the docstring for its `__init__` method
- ▶ Individual methods should be documented by their own docstring

# Operator Overloading

- ▶ Operator overloading lets you redefine the meaning of operators with respect to your own classes
- ▶ For example, operator overloading allow us to use the operator + to add two integers, as well as two string objects
  - ▶ i.e, the + operator is **overloaded** for the int class and the str class
- ▶ Operator overloading is achieved by defining a special method in the class definition
- ▶ The names of these methods start and end with double underscores (\_)
  - ▶ For example, both the int class and str class implement the \_\_add\_\_() method
  - ▶ These methods are not private since they have both leading underscores and trailing underscores.

```
x, y = 10, 20
```

```
x.__add__(y) # the same as x + y
```

30

# Operator Overloading

- The following table lists the operators and their corresponding special methods:

| <b>Operator</b> | <b>Special Method</b>                  | <b>Description</b>       |
|-----------------|----------------------------------------|--------------------------|
| +               | <code>__add__(self, other)</code>      | Addition                 |
| -               | <code>__sub__(self, other)</code>      | Subtraction              |
| *               | <code>__mul__(self, other)</code>      | Multiplication           |
| /               | <code>__truediv__(self, other)</code>  | Division                 |
| //              | <code>__floordiv__(self, other)</code> | Integer Division         |
| %               | <code>__mod__(self, other)</code>      | Modulus                  |
| **              | <code>__pow__(self, other)</code>      | Exponentiation           |
| ==              | <code>__eq__(self, other)</code>       | Equal to                 |
| !=              | <code>__ne__(self, other)</code>       | Not equal to             |
| >               | <code>__gt__(self, other)</code>       | Greater than             |
| >=              | <code>__lt__(self, other)</code>       | Greater than or equal to |

| <b>Operator</b> | <b>Special Method</b>                  | <b>Description</b>        |
|-----------------|----------------------------------------|---------------------------|
| <               | <code>__lt__(self, other)</code>       | Less than                 |
| <=              | <code>__le__(self, other)</code>       | Less than or equal to     |
| in              | <code>__contains__(self, value)</code> | Membership operator       |
| [index]         | <code>__getitem__(self, index)</code>  | Index operator            |
| len()           | <code>__len__(self)</code>             | Number of elements        |
| str()           | <code>__str__(self)</code>             | The string representation |

# Operator Overloading

- ▶ The following Point class shows how you can overload operators in a class:

```
point.py
import math
class Point:
 def __init__(self, x = 0, y = 0):
 self.__x = x
 self.__y = y

 def get_x(self):
 return self.__x

 def set_x(self, x):
 self.__x = x

 def get_y(self):
 return self.__y

 def set_y(self, y):
 self.__y = y

 def dist_from_origin(self):
 return math.sqrt(self.__x ** 2 + self.__y * 2)
```

# Operator Overloading

```
def __add__(self, other_point):
 return Point(self.__x + other_point.__x, self.__y + other_point.__y)

def __sub__(self, other_point):
 return Point(self.__x - other_point.__x, self.__y - other_point.__y)

def __eq__(self, other_point):
 return self.__x == other_point.__x and self.__y == other_point.__y

def __gt__(self, other_point):
 return self.dist_from_origin() > other_point.dist_from_origin()

def __ge__(self, other_point):
 return self.dist_from_origin() >= other_point.dist_from_origin()

def __lt__(self, other_point):
 return self.dist_from_origin() < other_point.dist_from_origin()

def __le__(self, other_point):
 return self.dist_from_origin() <= other_point.dist_from_origin()

def __str__(self):
 return "({}, {})".format(self.__x, self.__y)
```

# Operator Overloading

- ▶ Testing the class in the main script:

```
p1 = Point(4, 7)
p2 = Point(10, 6)

print("p1:", p1)
print("p2:", p2)
print("Is p1 == p2?", p1 == p2)
print("Is p1 > p2?", p1 > p2)
print("Is p1 >= p2?", p1 >= p2)
print("Is p1 < p2?", p1 < p2)
print("Is p1 <= p2?", p1 <= p2)

p3 = p1 + p2
p4 = p1 - p2

print()
print("p3:", p3)
print("p4:", p4)
```

## Output

```
p1: (4, 7)
p2: (10, 6)
Is p1 == p2 ? False
Is p1 > p2 ? False
Is p1 >= p2 ? False
Is p1 < p2 ? True
Is p1 <= p2 ? True

p3: (14, 13)
p4: (-6, 1)
```

# Compound Assignment Operators

- ▶ Special methods that start with `_i`, such as `__iadd__`, `__isub__`, implement the augmented arithmetic assignments (`+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`)
- ▶ These methods should attempt to do the operation in-place (modifying `self`) and return the result (which is typically `self`)
- ▶ For example, let's define the operator `+=` in the `Point` class:

```
def __iadd__(self, other_point):
 self.__x += other_point.__x
 self.__y += other_point.__y
 return self
```

```
p7 = Point(5, 2)
p8 = Point(3, 4)
print("p7:", p7)
print("p8:", p8)
p7 += p8
print("p7:", p7)
```

```
p7: (5, 2)
p8: (5, 2)
p7: (8, 6)
```

# String Representations

- ▶ The special method `__repr__()` is called by the `repr()` built-in function to compute the “official” string representation of an object
- ▶ This is typically used for debugging, so it is important that the representation is information-rich and unambiguous
- ▶ It should provide enough information that could be used to recreate an object with the same value, so `eval(repr(obj)) == obj`
  - ▶ `eval(expression)` is a built-in function that takes a string, evaluates it as a Python expression and returns the result of the evaluated expression
- ▶ `__repr__()` is intended to be unambiguous, while `__str__()` is intended to be readable
- ▶ If a class defines `__repr__()` but not `__str__()`, then `__repr__()` is also used when an “informal” string representation of instances of that class is required

# String Representations

- ▶ Example for the differences between `__str__` and `__repr__`:

```
import datetime
today = datetime.datetime.now()
```

```
str(today)
```

```
'2018-07-31 07:20:26.631666'
```

```
repr(today)
```

```
'datetime.datetime(2018, 7, 31, 7, 20, 26, 631666)'
```

```
print(today) # calls str(today)
```

```
2018-07-31 07:20:26.631666
```

```
today # calls repr(today)
```

```
datetime.datetime(2018, 7, 31, 7, 20, 26, 631666)
```

```
d = eval("datetime.datetime(2018, 7, 31, 7, 16, 41, 367383)")
d
```

```
datetime.datetime(2018, 7, 31, 7, 16, 41, 367383)
```

# String Representations

- Let's implement the `__repr__` method in our Point class:

```
def __repr__(self):
 return "Point({}, {})".format(self._x, self._y)
```

```
p9 = Point(1, 2)
print("p9:", p9)
s = repr(p9)
print(s)
p10 = eval(s)
print("p10:", p10)
```

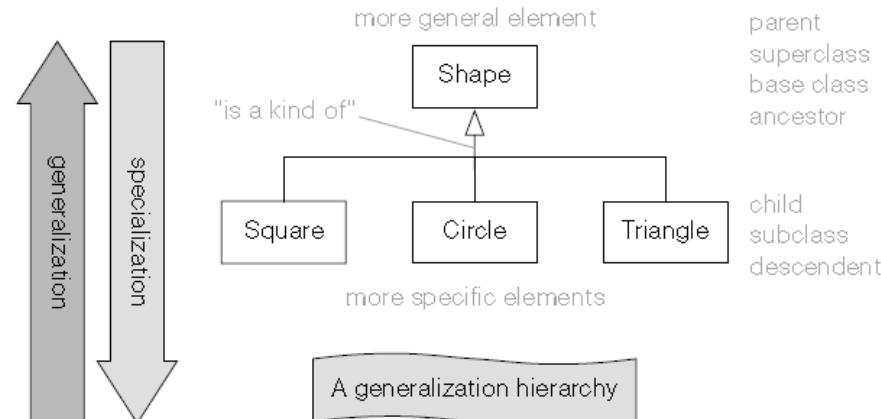
```
p9: (1, 2)
Point(1, 2)
p10: (1, 2)
```

## Exercise

- ▶ Create a class named Rational that represents a rational number
  - ▶ a rational number is any number that can be expressed as the fraction  $p/q$  of two integers: a numerator  $p$  and a non-zero denominator  $q$
- ▶ The class should support the following operators:
  - ▶ Addition, subtraction, multiplication, and division of two rationals
  - ▶ Adding an integer to a rational (both  $x + p/q$  and  $p/q + x$  should be supported)
  - ▶ The compound assignment operator  $+=$
  - ▶ Checking if two rationals are equal
    - ▶ Note that  $1/2 = 2/4 = 4/8$ , etc.
- ▶ Provide both informal and formal string representations for Rational

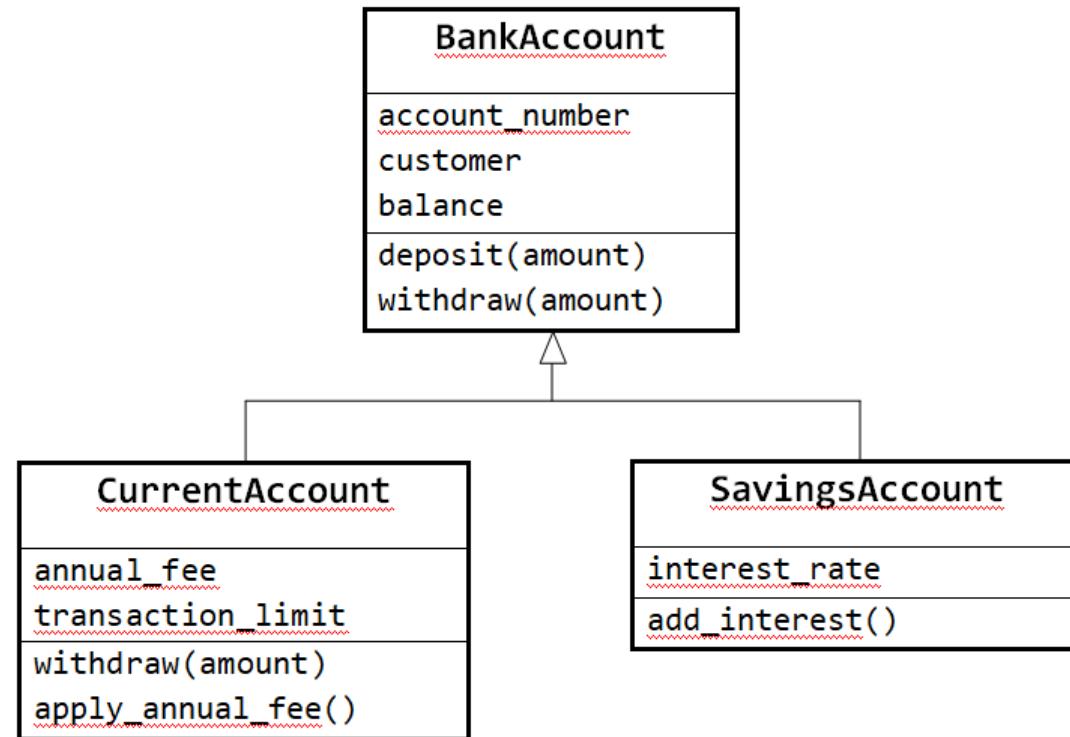
# Class Inheritance

- ▶ In OOP, **inheritance** enables new objects to take on the properties of existing objects
- ▶ A class that is used as the basis for inheritance is called a **superclass** or **base class**
- ▶ A class that inherits from a superclass is called a **subclass** or **derived class**
- ▶ Typically, a general type of object is defined by a base class, and then customized classes with more specialized functionality are derived from it
- ▶ The UML graphical representation of generalization is a hollow triangle shape on the superclass end of the line (or tree of lines) that connects it to one or more subtypes



# Class Inheritance

- ▶ We'll create the following class hierarchy:



# Class Inheritance in Python

- ▶ The syntax for a derived class definition looks like this:

```
class DerivedClassName(BaseClass):
 <statement-1>
 ...
 <statement-N>
```

- ▶ They can be defined in the same file that defines BankAccount or in a different Python file which imports BankAccount.

# Class Inheritance Demo

- Let's start with the SavingsAccount class:

```
from bank_account import BankAccount

class SavingsAccount(BankAccount):
 """A class representing a savings account"""

 def __init__(self, customer, account_number, interest_rate, balance=0):
 """Initialize the savings account"""
 super().__init__(customer, account_number, balance)
 self.interest_rate = interest_rate

 def add_interest(self):
 """Add interest to the account at the rate self.interest_rate"""
 self.balance *= (1 + self.interest_rate / 100)
```

SavingsAccount overrides the `__init__` method to allow the `interest_rate` to be initialized

The new `__init__` method calls the base class's `__init__` method in order to set the other attributes

The SavingsAccount class adds a new attribute, `interest_rate`, and a new method, `add_interest()` to its base class

The built-in function `super()` allows us to refer to the parent base class.

This is useful for accessing inherited methods that have been overridden in a class.

# Class Inheritance Demo

- ▶ Our new SavingsAccount might be used as follows:

```
from savings_account import SavingsAccount
```

```
my_savings = SavingsAccount("Matthew Walsh", 41522887, 3.5, 1000)
```

```
my_savings.check_balance()
```

```
The balance of account number 41522887 is $1000.00
```

```
my_savings.add_interest()
```

```
my_savings.check_balance()
```

```
The balance of account number 41522887 is $1035.00
```

# Class Inheritance Demo

- ▶ The second subclass, CurrentAccount, has a similar structure:

```
from bank_account import BankAccount

class CurrentAccount(BankAccount):
 """A class representing a current (checking) account"""

 def __init__(self, customer, account_number, annual_fee,
 transaction_limit, balance=0):
 """Initialize the current account"""
 super().__init__(customer, account_number, balance)
 self.annual_fee = annual_fee
 self.transaction_limit = transaction_limit

 def apply_annual_fee(self):
 """Deduct the annual fee from the account balance."""
 self.balance = max(0, self.balance - self.annual_fee)
```

# Class Inheritance Demo

```
def withdraw(self, amount):
 """Withdraw amount if sufficient funds exist in the account
 and amount is less than the transaction limit"""
 if amount <= 0:
 print("Invalid withdrawal amount")
 return

 if amount > self.balance:
 print("Insufficient funds")
 return

 if amount > self.transaction_limit:
 print(f"{self.currency}{amount:.2f} exceeds the transaction limit of "
 f"{self.currency}{self.transaction_limit:.2f}")
 return

 self.balance -= amount
```

# Class Inheritance Demo

- ▶ Note what happens if we call withdraw on a CurrentAccount object:

```
from current_account import CurrentAccount
```

```
my_current = CurrentAccount("Alison Wicks", 78300991, 20, 200)
```

```
my_current.withdraw(220)
```

Insufficient funds

```
my_current.deposit(750)
my_current.check_balance()
```

The balance of account number 78300991 is \$750.00

```
my_current.withdraw(220)
```

\$220.00 exceeds the transaction limit of \$200.00

- ▶ The withdraw method called is that of the CurrentAccount class, as this method overrides that of the same name in the base class, BankAccount

# The type() Function

- ▶ The `type(object)` built-in function returns the type of an *object*
- ▶ It is essentially the same as `object.__class__`
- ▶ Examples:

```
type(my_savings)
```

```
savings_account.SavingsAccount
```

```
type("hello")
```

```
str
```

## isinstance()

- ▶ The **isinstance(object, classinfo)** returns true if the *object* argument is an instance of the *classinfo* argument, or a subclass of it
- ▶ This function is recommended for testing the type of an object, because it takes subclasses into account
- ▶ Examples:

```
isinstance(my_savings, SavingsAccount)
```

True

```
isinstance(my_savings, BankAccount)
```

True

```
isinstance(my_savings, CurrentAccount)
```

False

## issubclass()

- ▶ A natural complement to `isinstance()` is the ability to determine whether one class has another class somewhere in its inheritance chain
- ▶ The `issubclass(class, classinfo)` returns true if *class* is a subclass of *classinfo*
- ▶ A class is considered a subclass of itself

```
issubclass(SavingsAccount, BankAccount)
```

True

```
issubclass(int, object)
```

True

- ▶ The following relationship between `isinstnace()` and `issubclass()` is always true:

```
isinstance(obj, cls) == issubclass(type(obj), cls)
```

## Exercise

- ▶ Write the class `VehicleForSale` as a subclass of `Vehicle`
- ▶ In addition to the attributes that are declared in the base class, `VehicleForSale` should have a depreciation value
- ▶ It should also have a method called `setDepreciation()` that accepts the percentage rate, calculates the depreciation value (which is `rate * price of vehicle`) and set this attribute on the object
- ▶ Override the `__str__` method that formats its returned value in the following manner:

# Abstract Base Classes

- ▶ **Abstract classes** are classes that contain one or more abstract methods
- ▶ **An abstract method** is a method that is declared, but contains no implementation
- ▶ Abstract classes cannot be instantiated, and require subclasses to provide implementations for the abstract methods
- ▶ A class that is derived from an abstract class cannot be instantiated, unless all of its abstract methods are overridden
- ▶ The module **abc** provides the infrastructure for defining **Abstract Base Classes (ABCs)**
- ▶ In the following example, we'll define the class Shape as an abstract base class

# Abstract Base Classes

## ► The Shape base class:

```
from abc import ABC, abstractmethod

class Shape(ABC):
 def __init__(self, location, color):
 self.location = location # of type Point
 self.color = color

 def move(self, delta_x, delta_y):
 self.location.x += delta_x
 self.location.y += delta_y

 @abstractmethod
 def get_area(self):
 pass

 @abstractmethod
 def display(self):
 print(f"Location: ({self.location.x}, {self.location.y})")
 print(f"Color: {self.color}")
 print(f"Area: {self.get_area():.3f}")
```

An abstract method can provide some basic functionality in the abstract class.  
Subclasses will still be forced to override the method's implementation.  
However, they will be able to invoke the abstract method using super().  


# Abstract Base Classes

- ▶ Trying to create a Shape object will raise an exception that Shape can't be instantiated:

```
from point import Point
from shape import Shape
```

```
s = Shape(Point(2, 5), "blue")
```

```

TypeError Traceback (most recent call last)
<ipython-input-2-15403aa0c3b5> in <module>()
----> 1 s = Shape(Point(2, 5), "blue")
```

```
TypeError: Can't instantiate abstract class Shape with abstract methods draw,
get_area
```

# Abstract Base Classes

- ▶ We'll now define a Circle subclass of Shape
- ▶ Circle overrides Shape's abstract methods, which turns it into a **concrete class** (a class that can be instantiated)

```
from shape import Shape
import math

class Circle(Shape):
 def __init__(self, location, color, radius):
 super().__init__(location, color)
 self.radius = radius

 def get_area(self):
 return math.pi * self.radius ** 2

 def display(self):
 super().display()
 print(f"Radius: {self.radius}")
```

# Abstract Base Classes

- ▶ Now it is possible to create Circle objects:

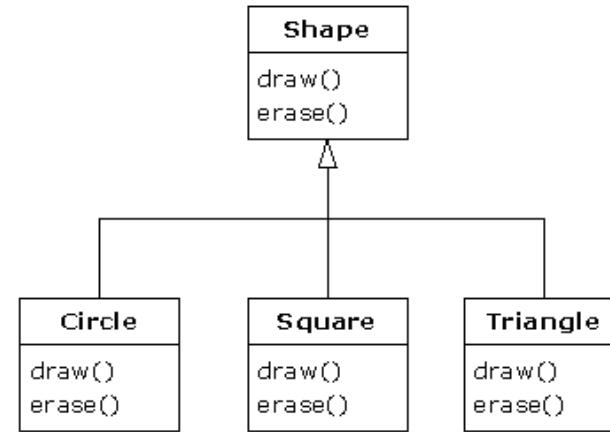
```
from circle import Circle
c = Circle(Point(3, 4), "red", 2)
```

```
c.display()
```

```
Location: (3, 4)
Color: red
Area: 12.566
Radius: 2
```

# Polymorphism

- ▶ **Polymorphism** is the ability of objects of different types to respond, each in its own way, to identical messages (method calls)
- ▶ For example, all shapes will draw when calling their draw() method, but each draw() is implemented in a different way



- ▶ Polymorphism allows for flexibility and loose coupling so that code can be extended and easily maintained over time

## Exercise

- ▶ Write a program that gets an input file of zoo animal data and interactively answers queries from a user
- ▶ The first line of each animal data record in the input file is in the following format:

```
Name AnimalType Species Mass
```

- ▶ Name: The animal's name. Contains no white space.
- ▶ AnimalType: The type of animal. One of: Mammal, Reptile, Bird.
- ▶ Species: The animal's species. Contains no white space.
- ▶ Mass: The animal's mass (weight) in kgs. An integer number  $\geq 1$ .
- ▶ Here are examples of first lines of animal data records:

```
Bob Mammal Bear 150
Lucy Reptile Lizard 1
Oliver Bird Ostrich 35
```

## Exercise

- ▶ The data following the first line in each animal record depends on the AnimalType
- ▶ Mammals:
  - ▶ The first line of data for animals of type Mammal is followed by: LitterSize, which is the average number of offspring the mammal has (an integer number  $\geq 1$  )
- ▶ Reptiles:
  - ▶ The first line of data for animals of type Reptile is followed by: VenomousOrNot, which indicates whether the reptile has a venomous bite
- ▶ Birds:
  - ▶ The first line of data for animals of type Bird is followed by: Wingspan (the wingspan of the bird in centimeters), and TalksOrMute (indicates whether the bird talks)
  - ▶ Birds that talk have an additional line of data: Phrase, which is what the bird says
    - ▶ May contain whitespace, but is no more than one line long.

# Exercise

- ▶ Here is an example of a data file to be read by the program:

```
Bob Mammal Bear 300
2
Lucy Reptile Lizard 2
Nonvenomous
Carl Reptile Cottonmouth 3
Venomous
Oliver Bird Ostrich 75
60 Mute
Polly Bird Parrot 1
2 Talks
I want a cracker
Doug Mammal Dog 20
4
```

## Exercise

- ▶ Once the program has read in the data file, it should request and process interactive queries from the user
- ▶ It should request queries in the following format:
- ▶ Query animal species[s], mass[m], litter[l], venom[v], wingspan[w], talk[t] or exit session[e]?
- ▶ The following example demonstrates all varieties of queries and responses, using the data provided in the above example file
- ▶ The program should gracefully handle cases in which a user does not respond with one of s, m, l, v, w, t, or e to the program's query request, or provides the name of an animal not in the database

# Exercise

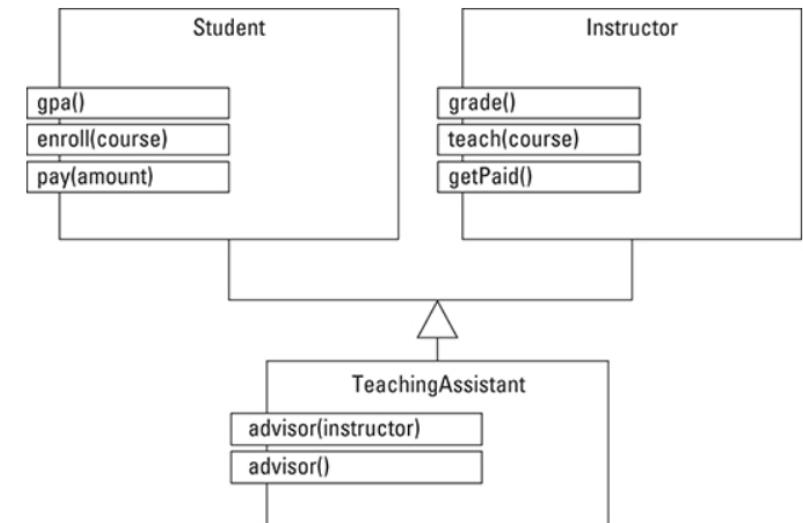
```
Query animal species[s], mass[m], litter[l], venom[v], wingspan[w], talk[t] or exit session[e]? s
Animal Name? Bob
Bob species is Bear
Query animal species[s], mass[m], litter[l], venom[v], wingspan[w], talk[t] or exit session[e]? m
Animal Name? Lucy
Lucy mass is 2
Query animal species[s], mass[m], litter[l], venom[v], wingspan[w], talk[t] or exit session[e]? l
Animal Name? Bob
Bob litter size is 2
Query animal species[s], mass[m], litter[l], venom[v], wingspan[w], talk[t] or exit session[e]? v
Animal Name? Carl
Carl is Venomous
Query animal species[s], mass[m], litter[l], venom[v], wingspan[w], talk[t] or exit session[e]? v
Animal Name? Lucy
Lucy is Nonvenomous
Query animal species[s], mass[m], litter[l], venom[v], wingspan[w], talk[t] or exit session[e]? w
Animal Name? Oliver
Oliver Wing Span is 60
Query animal species[s], mass[m], litter[l], venom[v], wingspan[w], talk[t] or exit session[e]? t
Animal Name? Oliver
Oliver is Mute
Query animal species[s], mass[m], litter[l], venom[v], wingspan[w], talk[t] or exit session[e]? t
Animal Name? Polly
Polly talks.
Polly says I want a cracker
Query animal species[s], mass[m], litter[l], venom[v], wingspan[w], talk[t] or exit session[e]? e
Goodbye!
```

# Multiple Inheritance

- ▶ Python also supports a form of multiple inheritance
- ▶ A class definition with multiple base classes looks like this:

```
class DerivedClassName(BaseClass1, BaseClass2, ...):
 <statement-1>
 ...
 <statement-N>
```

- ▶ For example, most universities offer jobs to graduate students as teaching assistants
- ▶ These individuals are still students and have all the properties of a student — but, in addition, they are also afforded the features of an instructor
- ▶ Thus, logically, their inheritance tree looks like this:



# Multiple Inheritance

- ▶ Let's build this hierarchy in Python
- ▶ We'll start with the Student class definition:

```
class Student:
 """A class representing a student"""\n def __init__(self, id, name):
 self.id = id
 self.name = name
 self.grades = []\n\n def add_grade(self, grade):
 self.grades.append(grade)\n\n def grades_average(self):
 return sum(self.grades) / len(self.grades)
```

# Multiple Inheritance

- ▶ We now define the Instructor class:

```
class Instructor:
 """A class representing an instructor"""
 def __init__(self, id, name, salary):
 self.id = id
 self.name = name
 self.salary = salary
 self.courses = []

 def add_course(self, course):
 self.courses.append(course)

 def get_courses(self):
 return self.courses
```

# Multiple Inheritance

- ▶ Lastly, `TeachingAssistant` inherits both from “`Student`” and “`Instructor`”:

```
from student import Student
from instructor import Instructor

class TeachingAssistant(Student, Instructor):
 """A class representing a teaching assistant"""
 def __init__(self, id, name, salary, advisor):
 Student.__init__(self, id, name)
 Instructor.__init__(self, id, name, salary)
 self.advisor = advisor
```

Calling the `__init__()` method of  
both parents

# Multiple Inheritance

- ▶ A sample test:

```
from ta import TeachingAssistant

ta1 = TeachingAssistant(3358123, "John Smith", 10000, "Ellen DeGeneres")

ta1.add_grade(80)
ta1.add_grade(90)
ta1.grades_average()
```

85.0

```
ta1.add_course("Algebra")
ta1.add_course("Python")
ta1.add_course("C++")
ta1.get_courses()
```

['Algebra', 'Python', 'C++']

# Iterables and Iterators

- ▶ An object is considered iterable if it can yield objects one at a time, typically within a for loop
- ▶ In Python, an object is iterable if passing it into the `iter()` function returns an `iterator`
- ▶ Internally, `iter()` inspects the object passed in, looking for an `__iter__()` method
- ▶ If such a method is found, it's called without any arguments and is expected to return an iterator
- ▶ For example, since a list is iterable, calling `iter()` on a list returns its iterator:

```
list1 = iter([1, 2, 3])
list1

<list_iterator at 0x2141d5c3e10>
```

- ▶ Each time we call the `__next__` method on the iterator, it gives us the next element
- ▶ If there are no more elements, it raises a `StopIteration`

# Iterables and Iterators

```
it.__next__()
```

1

```
it.__next__()
```

2

```
it.__next__()
```

3

```
it.__next__()
```

```

StopIteration Traceback (most recent call last)
<ipython-input-9-74e64ed6c80d> in <module>()
----> 1 it.__next__()

StopIteration:
```

# Iterables and Iterators

- ▶ Iterators are implemented as classes
- ▶ The required interface consists of just two methods:
- ▶ **`__iter__()`** – iterators should be iterable on their own as well
  - ▶ Typically this method just returns self
- ▶ **`__next__()`** – this method returns a new value for the next pass in the loop
  - ▶ When there are no more items to provide, the method should raise a `StopIteration` exception
  - ▶ When this is raised, the loop is considered complete, and execution resumes on the next line after the end of the loop

# Iterables and Iterators

- ▶ For example, let's create a class Range which provides a similar functionality as the built-in range() function

```
class Range:
 def __init__(self, count):
 self.count = count

 def __iter__(self):
 return RangeIter(self.count)

class RangeIter:
 def __init__(self, count):
 self.count = count
 self.current = 0

 def __iter__(self):
 return self

 def __next__(self):
 value = self.current
 self.current += 1
 if self.current > self.count:
 raise StopIteration
 return value
```

# Iterables and Iterators

- ▶ We can now use the Range class as the built-in range() function:

```
from myrange import Range
```

```
r = Range(5)
```

```
list(r)
```

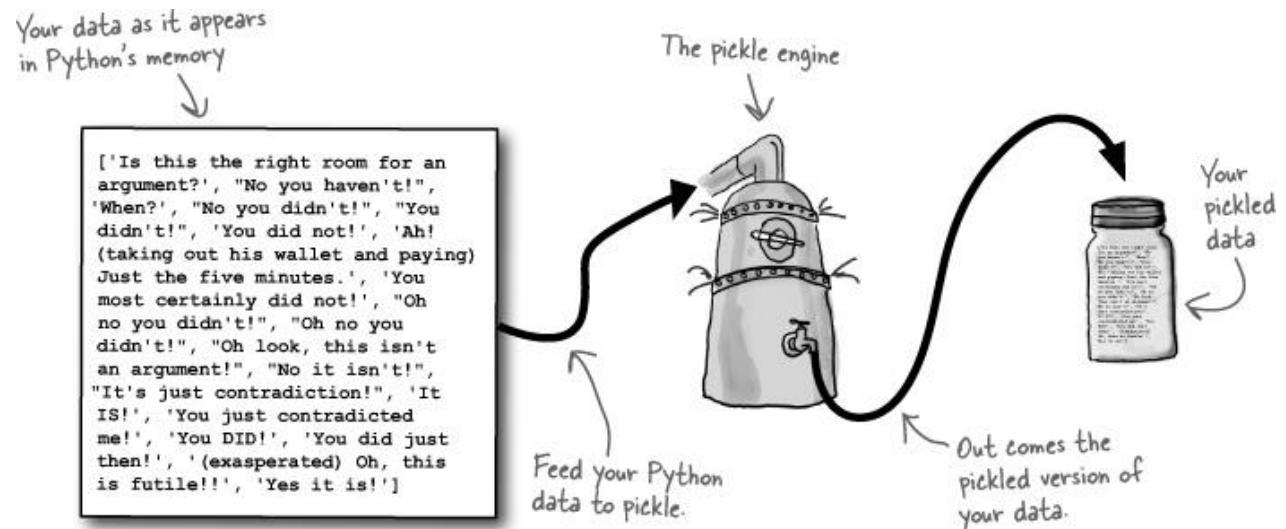
```
[0, 1, 2, 3, 4]
```

```
for i in r:
 print(i, end=" ")
```

```
0 1 2 3 4
```

# Pickling

- ▶ Pickling is the name of the object serialization process in Python
- ▶ By pickling we can convert an object hierarchy to a binary format that can be stored
- ▶ Unpickling is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy



# Pickling Objects

- ▶ To pickle an object we just need to import the pickle module and call the **dumps()** function passing the object to be pickled as a parameter:

```
import pickle
```

```
pickle.dumps(1)
```

```
b'\x80\x03K\x01.'
```

```
pickle.dumps([1, 2, 3])
```

```
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

- ▶ The pickled output contains more information than the original value, because it also needs to store the type, so the object can be reconstituted later
- ▶ To store the pickled object in a file, call the **dump()** function, which also takes an opened binary file in which the pickling result will be stored automatically

```
with open("mydata.bin", "wb") as my_data:
 pickle.dump([1, 2, 3], my_data)
```

# Unpickling Objects

- ▶ The unpickling process is done by using the **load()** or **loads()** functions of the pickle module, which return a complete object hierarchy from a simple bytes array
- ▶ The difference between the two is similar to the dump functions: **load()** accepts a readable file object, while **loads()** accepts a string

```
pickled = pickle.dumps(1)
pickled
```

```
b'\x80\x03K\x01.'
```

```
pickle.loads(pickled)
```

```
1
```

```
with open("mydata.bin", "rb") as my_data:
 my_list = pickle.load(my_data)
print(my_list)
```

```
[1, 2, 3]
```

- ▶ The same pickling and unpickling process can be applied to your own objects

# Unpicklable Objects

- ▶ Not every object is pickleable
- ▶ Some objects like db connections and handles to opened files can't be pickled, and trying to pickle them raises a `pickle.PickleError` exception
- ▶ If there are unpicklable objects in the hierarchy of the object you want to pickle, this prevents you from serializing (and storing) the entire object
- ▶ Other attributes that should avoid being pickled are initialization values, system-specific details, and other transient information which is not part of the object's value directly
- ▶ Fortunately, Python offers you a way to specify what you want to pickle and how to unpickle

# Customizing Pickling

- ▶ Python provides two special methods to control how individual objects are pickled and restored
- ▶ **`__getstate__()`** controls what gets included in the pickled value
  - ▶ For complex objects, the value will typically be a dictionary or a tuple
  - ▶ If `__getstate__()` is absent, the instance's `__dict__` is pickled as usual
- ▶ **`__setstate__(state)`** is called upon unpickling, and accepts the state of the object to restore
  - ▶ The state is exactly the same value that was returned from `__getstate__()`
  - ▶ If `__setstate__()` is absent, the pickled state must be a dictionary and its items are assigned to the new instance's dictionary
- ▶ If `__getstate__()` returns a false value, the `__setstate__(state)` method will not be called upon unpickling

# Customizing Pickling Example

---

- ▶ For example, a currency conversion class might contain the current amount as well as a string to indicate the currency being represented
- ▶ In addition, it would likely have access to a dictionary of current exchange rates, so that it can convert the amount to a different currency
- ▶ Because the currency conversion values aren't specific to the instance at hand, and they'll change over time anyway, there's no reason to store them in the pickled string
- ▶ Thus, we can use `__getstate__()` to provide just those values that are actually important

# Customizing Pickling Example

```
class Money:
 def __init__(self, amount, currency):
 self.amount = amount
 self.currency = currency
 self.conversion = { 'USD': 1, 'CAD': 1.31, 'EUR': 0.87 }

 def convert(self, currency):
 ratio = self.conversion[currency] / self.conversion[self.currency]
 return Money(self.amount * ratio, currency)

 def __str__():
 return f"{self.amount:.2f} {self.currency}"

 def __repr__():
 return f"Money({self.amount} {self.currency})"

 def __getstate__():
 return self.amount, self.currency

 def __setstate__(self, state):
 self.amount = state[0]
 self.currency = state[1]
```

# Customizing Pickling Example

## ▶ Sample usage:

```
from money import Money
```

```
my_money = Money(250, "USD")
print(my_money)
print(my_money.convert("EUR"))
```

250.00 USD

217.50 EUR

```
with open("money.bin", "wb") as my_file:
 pickle.dump(my_money, my_file)
```

```
with open("money.bin", "rb") as my_file:
 saved_money = pickle.load(my_file)
```

```
print(saved_money)
```

250.00 USD

# Modules

- ▶ As we've seen, Python is quite a modular language and has functionality beyond the core programming essentials
- ▶ In programming, a **module** is a piece of software that has a specific functionality
- ▶ Modules are a good way to organize large programs in a hierarchical way
- ▶ For example, when building a game, one module would be responsible for the game logic, another module would be responsible for drawing the game on the screen, etc.
- ▶ Each module is written in a different file, which can be edited separately

# Python Modules

- ▶ A module in Python is simply a Python file that ends with .py extension
- ▶ The name of the module will be the name of the file
  - ▶ For example, a Python file called hello.py has the module name of hello that can be imported into other Python files or used on the Python command line interpreter
- ▶ Modules can define functions, classes, and variables that you can reference in other Python files
- ▶ In the following example we will create two modules in the folder mygame:

```
mygame/
 game.py
 ui.py
```

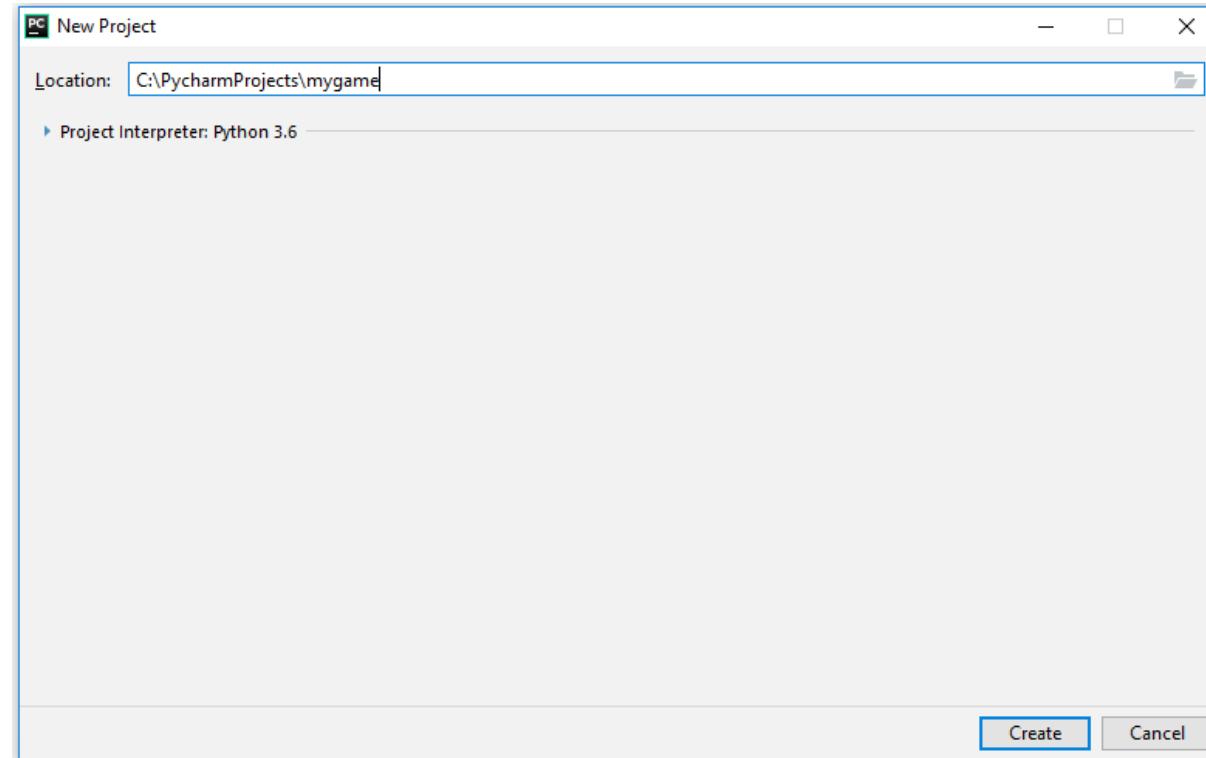
- ▶ The Python script game.py implements the game logic
- ▶ It will use functions from ui.py, which is responsible for drawing the game on screen and interacting with the user

# Module Names

- ▶ Note that because of the syntax of the import statement, you should avoid naming your module anything that isn't a valid Python identifier
- ▶ For example, the filename <module>.py should not contain a hyphen or start with a digit
- ▶ Do not give your module the same name as any built-in modules (such as math or random) because these get priority when Python imports

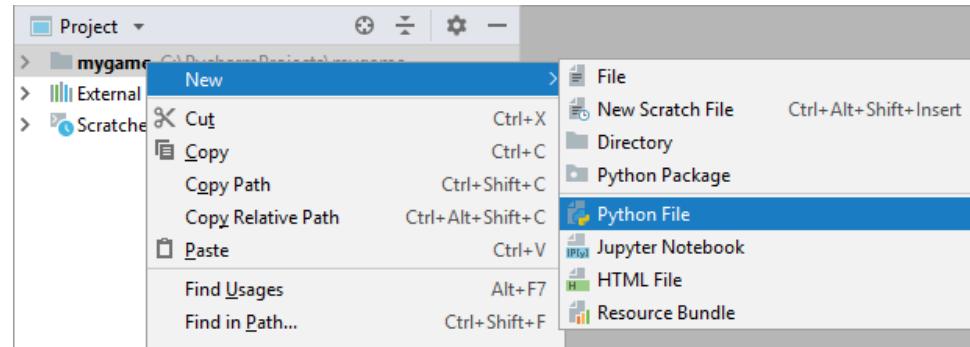
# Create a New PyCharm Project

- ▶ Create a new project called mygame in PyCharm:

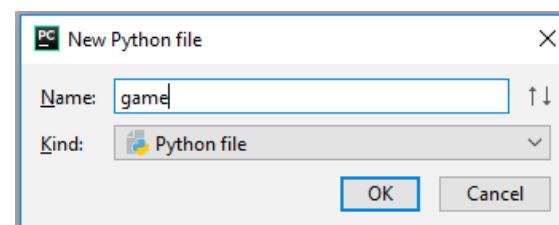


# Creating a Module

- ▶ Add a new Python script to the project, by right-clicking the project name in the Project pane and choosing New -> Python File



- ▶ Name the file game.py



- ▶ Similarly, add another module named ui.py

# Creating a Module

- ▶ Enter the following code in ui.py:

```
ui.py

def print_welcome_message():
 print("Welcome to my amazing game!")
 print("Press ENTER to start")
 input()

def draw_board():
 print("====")
 print("====")
 print("==== Game board ====")
 print("====")
 print("====")
 print()

def get_move(player_num):
 print("Player {}: it's your turn".format(player_num))
 move = input("Enter your move: ")
 return move
```

# Importing a Module

- ▶ To use functions, constants or classes defined inside a module, we first have to **import** it using the import statement
- ▶ The syntax of the import statement is:

```
import module_name
```

- ▶ Upon encountering the line `import <module>`, the Python interpreter:
  - ▶ Executes the statements in the file `<module>.py`
  - ▶ Enters the module name `<module>` into the current namespace, so that the attributes it defines are available with the “dotted syntax”: `<module>.<attribute>`

# Importing a Module

- ▶ Since game.py uses the functions defined in ui.py, it should first import it:

```
game.py

import the ui module
import ui

def start():
 ui.print_welcome_message()
 play_game()

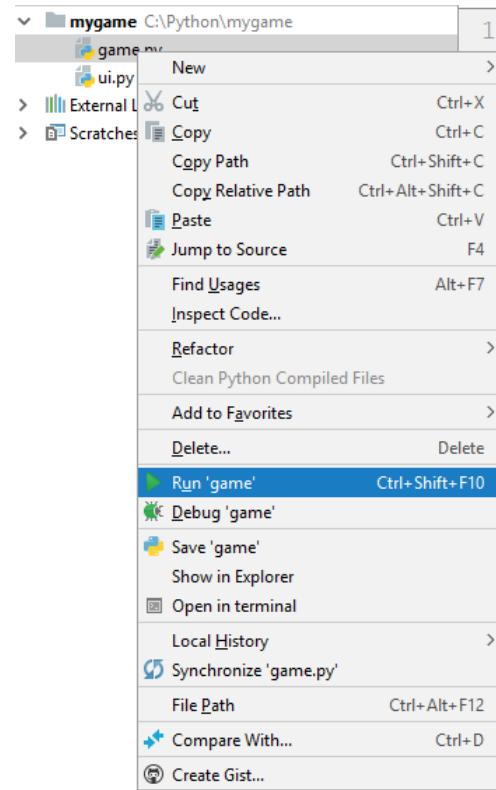
def play_game():
 current_player = 0
 move = None
 while move != "exit":
 ui.draw_board()
 move = ui.get_move(current_player)
 current_player = 1 - current_player

if __name__ == "__main__":
 start()
```

This means that main() will be executed only when the module is run directly, but not when it is imported by another module

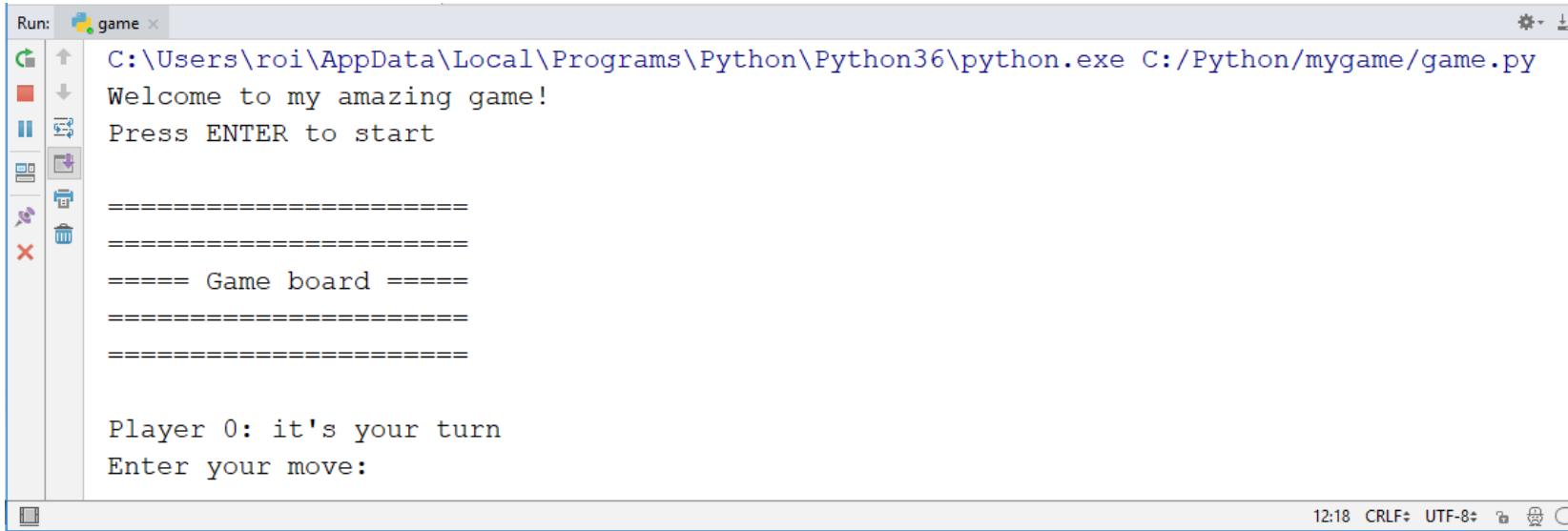
# Running a Module

- ▶ You can run a module by right-clicking its name in the Project pane and choosing Run... (or pressing Ctrl+Shift+F10)



# Running a Module

- ▶ A Run pane will open where you can see the program output:



The screenshot shows a Windows-style application window titled "Run: game". The window contains the following text output from a Python script:

```
C:\Users\roi\AppData\Local\Programs\Python\Python36\python.exe C:/Python/mygame/game.py
Welcome to my amazing game!
Press ENTER to start

=====
===== Game board =====
=====

Player 0: it's your turn
Enter your move:
```

The window has a toolbar on the left with icons for file operations like Open, Save, and Close, as well as other controls. The status bar at the bottom shows the time as 12:18 and encoding as UTF-8.

# Importing Selected Identifiers

- ▶ The statement **import my\_module** imports every identifier in the module
- ▶ To use only specific names from the module, use the following import statement:

```
from module_name import name1, name2, ..., nameN
```

- ▶ Any code after this import statement can use name1, name2, etc. without prefixing them with the module name

# Importing Selected Identifiers

- ▶ For example, game.py could import the specific functions it needs from ui.py:

```
game.py

from ui import print_welcome_message, draw_board, get_move

def start():
 print_welcome_message() ← No need to prefix the module name
 play_game()

def play_game():
 current_player = 0
 current_move = None

 while current_move != "exit":
 draw_board()
 current_move = get_move(current_player)
 current_player = 1 - current_player

if __name__ == "__main__":
 start()
```

No need to prefix the module name

# Importing All Identifiers

- ▶ You may also import every identifier from a module by using the `import *` command:

```
from module_name import *
```

- ▶ This statement is functionally equivalent to `import module_name`
- ▶ However, `import *` allows to access identifiers in the module without prefixing them with the module name
- ▶ By default, `import *` imports all identifiers in the module not starting with an underscore
  - ▶ You can override this behavior by assigning the variable `__all__` to a list of identifiers that shall be loaded and imported when `import *` is used
  - ▶ In general using `import *` is not recommended, since there is a danger of name conflicts (particularly if many modules are imported in this way)

# Importing All Identifiers

- ▶ For example, game.py could use import \* to import all identifiers from ui.py:

```
game.py

from ui import *

def start():
 print_welcome_message() ← No need to prefix the module name
 play_game()

def play_game():
 current_player = 0
 current_move = None

 while current_move != "exit":
 draw_board()
 current_move = get_move(current_player)
 current_player = 1 - current_player

if __name__ == "__main__":
 start()
```

# Aliasing Modules

- ▶ It is possible to modify the names of modules and their functions within Python by using the **as** keyword
- ▶ You may want to change a name because:
  - ▶ You've already used the same name for something else in your program
  - ▶ Another module you have imported also uses that name
  - ▶ You may want to abbreviate a longer name that you are using a lot
- ▶ The construction of this statement looks like this:

```
import module_name as another_name
```

- ▶ For example, we could abbreviate the name of the math module to m:

```
import math as m
```

```
print(m.pi)
print(m.e)
```

```
3.141592653589793
2.718281828459045
```

# Aliasing Modules

- ▶ You can also define an alias to a selected identifier imported from the module
- ▶ For example:

```
from math import radians as rad
```

```
print(rad(90))
```

```
1.5707963267948966
```

- ▶ For some modules, it is commonplace to use aliases
- ▶ For example, the matplotlib.pyplot module's official documentation calls for use of plt as an alias:

```
import matplotlib.pyplot as plt
```

# Module Initialization

- ▶ A module is loaded and initialized only once, regardless of the number of times you import the module in your script
- ▶ The first time a module is loaded into a running Python script, it is initialized by executing the code in the module
- ▶ If another module in your code imports the same module again, it will not be loaded again, so local variables inside the module act as a “singleton”

# Accessing Modules from Another Directory

- ▶ Modules may be useful for more than one project, and in that case it makes less sense to keep a module in a particular directory that's tied to a specific project
- ▶ The Python interpreter searches for the modules files in a list of directories that it receives from the **sys.path** variable
- ▶ If you want to use a Python module from a location other than the same directory where your main program is, you have a few options
  - ▶ Append the path of the module to the sys.path variable by calling `sys.path.append()`
  - ▶ Copy the module to your main system Python path, which will make the module available system-wide

# Accessing Modules from Another Directory

- ▶ For example, let's move ui.py to a different directory, e.g., C:\Python
- ▶ When trying to run game.py, we'll get the following error:

```
C:\Users\roi\Anaconda3\python.exe C:/PycharmProjects/mygame/game.py
Traceback (most recent call last):
 File "C:/PycharmProjects/mygame/game.py", line 4, in <module>
 import ui
ModuleNotFoundError: No module named 'ui'

Process finished with exit code 1
```

- ▶ To fix this error, you may append the path C:\Python to the system path *before* the import command:

```
import sys
sys.path.append(r"C:\Python")

import the ui module
import ui
...
```

# Accessing Modules from Another Directory

- ▶ Another option is to copy ui.py into the main system Python path
- ▶ To find out what path Python checks, run the Python interpreter, import the sys module, and print sys.path:

```
(base) C:\Users\roi>python
Python 3.6.4 |Anaconda, Inc.| (default, Jan 16 2018, 10:22:32) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> print(sys.path)
['', 'C:\\\\Users\\\\roi\\\\Anaconda3\\\\python36.zip', 'C:\\\\Users\\\\roi\\\\Anaconda3\\\\DLLs', 'C:\\\\Users\\\\roi\\\\Anaconda3\\\\lib', 'C:\\\\Users\\\\roi\\\\Anaconda3', 'C:\\\\Users\\\\roi\\\\AppData\\\\Roaming\\\\Python\\\\Python36\\\\site-packages', 'C:\\\\Users\\\\roi\\\\Anaconda3\\\\lib\\\\site-packages', 'C:\\\\Users\\\\roi\\\\Anaconda3\\\\lib\\\\site-packages\\\\win32', 'C:\\\\Users\\\\roi\\\\Anaconda3\\\\lib\\\\site-packages\\\\win32\\\\lib', 'C:\\\\Users\\\\roi\\\\Anaconda3\\\\lib\\\\site-packages\\\\Pythonwin']
```

- ▶ For example, we can copy ui.py to C:\Users\%UserName%\Anaconda3\Lib\site-packages
- ▶ This will make the module available system-wide
  - ▶ For example, you can use this module from the Python shell or IPython Notebook

## Exercise (27)

- ▶ Create a module `input_helper.py` that will provide functions for getting different types of inputs from the user, while handling any possible error in the input, and letting the user fix the errors
- ▶ The module should support the following operations:
  - ▶ Get an integer from the user
  - ▶ Get a positive integer from the user
  - ▶ Get an integer in a specified interval  $[a, b]$
  - ▶ Get a pair of two integers from the user (e.g., get the coordinates of a point)
- ▶ The functions should keep asking the user for input until he/she enters a valid one
- ▶ Make this module available system-wide
- ▶ Test the functions of this module from a Jupyter Notebook

# Exercise (27)

## ▶ Sample run:

```
import input_helper
```

```
x = input_helper.get_positive_int()
x
```

```
Enter a positive integer: hello
Incorrect input. Try again.
Enter a positive integer: -2
Number must be positive. Try again.
Enter a positive integer: 5
```

```
5
```

```
p = input_helper.get_pair()
p
```

```
Enter an integer: 8
Enter an integer: 6

(8, 6)
```

# Python Packages

- ▶ Packages are the natural way to organize and distribute larger Python projects
- ▶ A Python **package** is simply a structured arrangement of modules within a directory on the file system
- ▶ To make a package, the module files are placed in a directory, along with a file named `__init__.py`
- ▶ This file is run when the package is imported and may perform some initialization and its own imports
- ▶ It may be an empty file if no special initialization is required, but it must exist for the directory to be considered by Python to be a package
  - ▶ This is done to prevent directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path

# Python Packages

- ▶ For example, the NumPy package exists as the following directory:

```
numpy/
 __init__.py
 core/
 fft/
 __init__.py
 fftpack.py
 info.py
 ...
 linalg/
 __init__.py
 linalg.py
 info.py
 ...
 polynomial/
 __init__.py
 chebyshev.py
 legendre.py
 ...
 random/
 version.py
 ...
```

# Importing Packages

- ▶ For example, `polynomial` is a subpackage of the `numpy` package containing several modules, including `legendre`, which may be imported as

```
import numpy.polynomial.legendre
```

- ▶ To avoid having to use this full dotted syntax in actually referring to the module's attributes, it is convenient to use

```
from numpy.polynomial import legendre
```

# Creating a Package

- ▶ The steps to create a Python package are:
  - ▶ Create a directory and give it your package's name
  - ▶ Put your modules in it
  - ▶ Create a `__init__.py` file in the directory
- ▶ For example, let's create a package for our game
- ▶ First, we'll use the directory `C:\PycharmProjects\mygame` as our package directory
  - ▶ It already contains the modules `game.py` and `ui.py`
- ▶ Thus, we only need to create an empty file named `__init__.py` inside `mygame`
- ▶ Now copy the package to `C:\Users\%UserName%\Anaconda3\Lib\site-packages`
- ▶ The package is now available system-wide

# Testing the Package

- ▶ Create a new Jupyter notebook
- ▶ To use the module game, we can import it in two ways:

```
import mygame.game
```

```
mygame.game.start()
```

Welcome to my amazing game!  
Press ENTER to start

```
|
```

```
from mygame import game
```

```
game.start()
```

Welcome to my amazing game!  
Press ENTER to start

```
|
```

- ▶ In the first method, we must use the mygame prefix whenever we access the module game
- ▶ In the second method, we don't, because we import the module into our module's namespace

# Exporting Modules from a Package

- ▶ To make a module accessible directly from the package itself, you can import the module in the `__init__.py` file
- ▶ For example:

```
__init__.py
from . import game # relative import
```

- ▶ This imports the `game` module relative to the current package
- ▶ Now we only need to import the package in order to work with its modules:

```
import mygame
```

```
mygame.game.start()
```

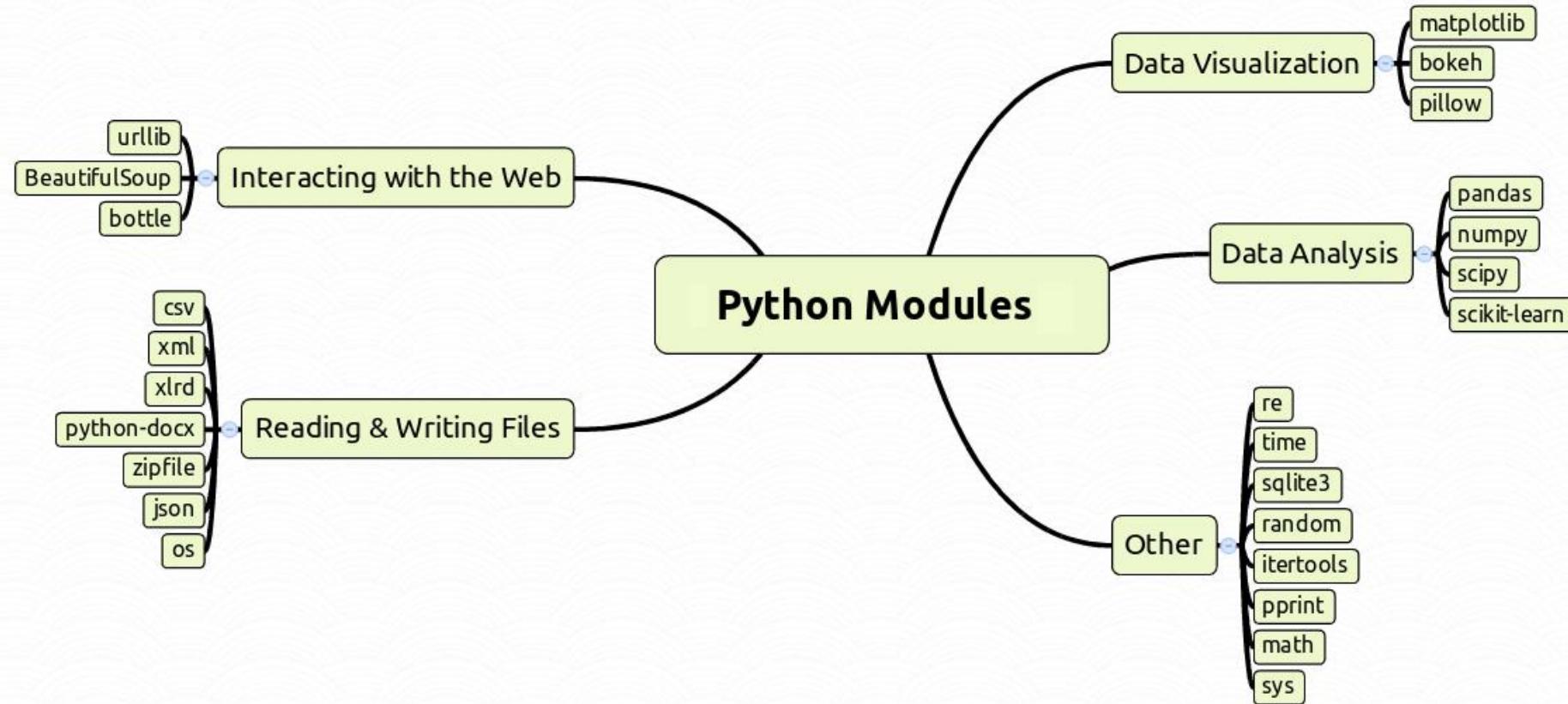
Welcome to my amazing game!  
Press ENTER to start

# Summary

---

- ▶ Use `import x` for importing packages and modules
- ▶ Use `from x import y` where x is the package prefix and y is the module name
- ▶ Use `import y as z` only when z is a standard abbreviation (e.g., np for numpy)
- ▶ Use `from x import y as z` if two modules named y are to be imported or if y is an inconveniently long name

# Python Built-in Modules/Packages



# Python Built-in Modules/Packages

- ▶ A list of some of the major, freely available Python modules and packages:

| Module/Package | Description                                                                          |
|----------------|--------------------------------------------------------------------------------------|
| os, sys        | Operating system services                                                            |
| math, cmath    | Mathematical functions                                                               |
| fractions      | Rational number arithmetic                                                           |
| random         | Random number generator                                                              |
| collections    | Data types for containers that extend the functionality of dictionaries, tuples, etc |
| itertools      | Tools for efficient iterators that extend the functionality of simple Python loops   |
| datetime       | Parsing and manipulating dates and times                                             |
| re             | Regular expressions                                                                  |
| argparse       | Parser for command line options and arguments                                        |
| urllib         | URL (including web pages) opening, reading and parsing                               |
| beautifulsoup  | HTML parser, with handling of malformed documents                                    |
| logging        | Python's built-in logging module                                                     |

# Python Built-in Modules/Packages

| <b>Module/Package</b> | <b>Description</b>                                                             |
|-----------------------|--------------------------------------------------------------------------------|
| unittest              | Unit testing framework for systematically testing and validating units of code |
| pdb                   | The Python debugger                                                            |
| xml, lxml             | XML parsers                                                                    |
| json                  | JSON parser                                                                    |
| pyodbc                | Python SQL driver                                                              |
| pymongo               | Python MongoDB driver                                                          |
| visual                | Three-dimensional visualization                                                |
| numpy                 | Numerical and scientific computing                                             |
| scipy                 | Scientific computing algorithms                                                |
| matplotlib            | Plotting                                                                       |
| pandas                | Data manipulation and analysis with table-like data structures                 |
| scikit-learn          | Machine learning                                                               |

# The sys Module

- ▶ The sys module provides some useful system-specific parameters and functions
- ▶ **sys.argv** is a list of strings, that holds the command line arguments passed to a Python program when it is executed
  - ▶ The first item, sys.argv[0], is the name of the program itself
- ▶ This allows for a degree of interactivity without having to read from configuration files or require direct user input,
- ▶ It also allows other programs or shell scripts to call a Python program and pass it particular input values or settings

# The sys Module

- ▶ For example, the following script squares a given number:

```
#square.py
import sys

num = int(sys.argv[1])
print(num, "squared is", num**2)
```

- ▶ Running this program from the command line with

```
python square.py 3
```

produces the following output:

```
C:\Python\sys>python square.py 3
3 squared is 9
```

- ▶ Because we did not hard-code the value of num, the same program can be run with any number

# The sys Module

- ▶ **sys.exit()** causes a program to terminate and exit from Python
- ▶ This happens “cleanly,” so that any commands specified in a try statement’s finally clause are executed first and any open files are closed
- ▶ The optional argument to `sys.exit()` can be any object:
  - ▶ If it is an integer, it is passed to the shell and indicates a success or failure of the program
    - ▶ 0 usually denotes “successful” termination of the program and nonzero values indicate some kind of error
  - ▶ Passing no argument or `None` is equivalent to 0
  - ▶ If it is a string, it is passed to `stderr` (the standard error stream)
    - ▶ Typically it appears as an error message on the console (unless redirected elsewhere by the shell)

# The sys Module

- ▶ A common way to help users with scripts that take command line arguments is to issue a usage message if they get it wrong, as in the following code example:

```
#square.py
import sys

try:
 num = int(sys.argv[1])
except (IndexError, ValueError):
 sys.exit("Please enter an integer, <n>, on the command line.\n"
 "Usage: python {} <n>".format(sys.argv[0]))

print(num, "squared is", num**2)
```

```
C:\Python\sys>python square.py
Please enter an integer, <n>, on the command line.
Usage: python square.py <n>

C:\Python\sys>python square.py 5
5 squared is 25
```

# The os Module

- ▶ The **os** module provides various operating system interfaces in a platform-independent way
- ▶ It provides a number of functions for retrieving information about the context in which the Python process is running
- ▶ For example, **os.uname()** returns information about the operating system running Python and the network name of the machine running the process
- ▶ **os.getenv(key)** returns the value of the environment variable key if it exists
- ▶ Commonly used environment variables:
  - ▶ HOME: the path to the user's home directory
  - ▶ PWD: the current working directory
  - ▶ USER: the current user's username and
  - ▶ PATH: the system path environment variable

```
print(os.getenv('HADOOP_HOME'))
```

C:\Hadoop

# File System Commands

- The os module also provides functions to navigate the system directory tree and manipulate files and directories from within a Python program

| Function                                   | Description                                                                                                                                                  |
|--------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>os.listdir(path=':')</code>          | List the entries in the directory given by <i>path</i> (or the current working directory if this is not specified)                                           |
| <code>os.remove(path)</code>               | Delete the file <i>path</i> (raises an OSError if path is a directory; use <code>os.rmdir</code> instead)                                                    |
| <code>os.rmdir(path)</code>                | Delete the directory <i>path</i> . If the directory is not empty, an OSError is raised.                                                                      |
| <code>os.rename(old_name, new_name)</code> | Rename the file or directory <i>old_name</i> to <i>new_name</i> . If a file with the name <i>new_name</i> already exists, it will be overwritten.            |
| <code>os.mkdir(path)</code>                | Create the directory named <i>path</i>                                                                                                                       |
| <code>os.system(command)</code>            | Execute <i>command</i> in a subshell. If the command generates any output, it is redirected to the interpreter standard output stream, <code>stdout</code> . |

# File System Commands

- ▶ Example for listing all the files in a user-supplied folder:

```
dir_name = input("Enter a path: ")
for filename in os.listdir(dir_name):
 print(filename)
```

```
Enter a path: C:\Notebooks\pandas
.ipynb_checkpoints
DataFrame.ipynb
DataFrameBasicsEx.ipynb
DescriptiveStatistics.ipynb
DescriptiveStatisticsEx.ipynb
euro_winners.csv
Indexing.ipynb
IndexingEx.ipynb
```

# Pathname Manipulations

- ▶ `os.path` module provides a number of useful functions for manipulating pathnames:

| Function                                     | Description                                                                                                                                                             |
|----------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>os.path.basename(path)</code>          | Return the basename of the pathname <i>path</i> giving a relative or absolute path to the file: this usually means the filename.                                        |
| <code>os.path.dirname(path)</code>           | Return the directory of the pathname <i>path</i>                                                                                                                        |
| <code>os.path.split(path)</code>             | Split <i>path</i> into a directory and a filename, returned as a tuple (equivalent to calling dirname and basename) respectively                                        |
| <code>os.path.splitext(path)</code>          | Split <i>path</i> into a ‘root’ and an ‘extension’ (returned as a tuple pair)                                                                                           |
| <code>os.path.exists(path)</code>            | Return True if the directory or file path exists, and False otherwise                                                                                                   |
| <code>os.path.getmtime(path)</code>          | Return the time of last modification of <i>path</i>                                                                                                                     |
| <code>os.path.getsize(path)</code>           | Return the size of <i>path</i> in bytes                                                                                                                                 |
| <code>os.path.join(path1, path2, ...)</code> | Return a pathname formed by joining the path components <i>path1</i> , <i>path2</i> , etc. with the directory separator appropriate to the operating system being used. |

# Pathname Manipulations

- ▶ Some examples referring to a file C:\Notebooks\Test.ipynb:

```
os.path.basename(r'C:\Notebooks\test.ipynb') # just the filename

'test.ipynb'
```

```
os.path.dirname(r'C:\Notebooks\test.ipynb') # just the directory

'C:\\Notebooks'
```

```
Directory and filename in a tuple
os.path.split(r'C:\Notebooks\test.ipynb')

('C:\\Notebooks', 'test.ipynb')
```

```
File path stem and extension in a tuple
os.path.splitext(r'C:\Notebooks\test.ipynb')

('C:\\Notebooks\\test', '.ipynb')
```

```
Join directories and/or file name
os.path.join(r'C:\Notebooks', 'test.ipynb')
```

```
'C:\\Notebooks\\test.ipynb'
```

```
os.path.exists(r'C:\Notebooks\test.py') # file does not exist

False
```

## Exercise (28)

- ▶ Suppose you have a directory of data files identified by filenames containing a date in the form data-DD-Mon-YY.txt
  - ▶ DD is the two-digit day number, Mon is the three-letter month abbreviation and YY is the last two digits of the year, for example '02-Feb-18'
- ▶ Write a program that converts the filenames into the form data-YYYY-MM-DD.txt so that an alphanumeric ordering of the filenames puts them in chronological order

# The random Module

- ▶ For simulations, modeling and some numerical algorithms it is often necessary to generate random numbers from some distribution
- ▶ Python, as many other programming languages, implements a **pseudorandom number generator (PRNG)** – an algorithm that generates a sequence of numbers that approximates the properties of “truly” random numbers
- ▶ Such sequences are determined by an **originating seed** and are always the same following the same seed
  - ▶ This is good for reproducing calculations involving random numbers, but a bad thing if used for cryptography, where the random sequence must be kept secret
- ▶ Any PRNG will yield a sequence that eventually repeats
  - ▶ A good generator will have a long period
- ▶ The PRNG implemented by Python is the Mersenne Twister
  - ▶ A well-respected and much-studied algorithm with a period of  $2^{19937} - 1$

# Generating Random Numbers

- ▶ The random number generator can be seeded with any *hashable object*
  - ▶ e.g., an immutable object such as an integer
- ▶ When the random module is first imported, it is seeded with a representation of the current system time
  - ▶ unless the operating system provides a better source of a random seed
- ▶ The PRNG can be reseeded at any time with a call to **random.seed()**
- ▶ The basic random number method is **random.random()**
  - ▶ It generates a random number selected from the uniform distribution in the semi-open interval  $[0, 1)$  – that is, including 0 but not including 1

# Generating Random Numbers

```
import random
random.random() # PRNG seeded "randomly"
```

0.22321073814882275

```
random.seed(42) # seed the PRNG with a fixed value
```

```
random.random()
```

0.6394267984578837

```
random.random()
```

0.025010755222666936

```
random.seed(42) # reseed with the same value as before
```

```
random.random()
```

0.6394267984578837

```
random.random()
```

0.025010755222666936

# Generating Random Numbers

- To select a random *floating point* number,  $N$ , from a given range,  $a \leq N \leq b$ , use **random.uniform( $a, b$ )**:

```
random.uniform(-2, 2)
```

```
-0.899882726523523
```

```
random.uniform(-2, 2)
```

```
-1.107157047404709
```

- To select a random *integer*,  $N$ , in a given range,  $a \leq N \leq b$ , use **random.randint( $a, b$ )**:

```
random.randint(5, 15)
```

```
6
```

```
random.randint(5, 15)
```

```
15
```

# Generating Random Numbers

- ▶ The random module has several methods for drawing random numbers from nonuniform distributions (see the [documentation](#))
- ▶ For example, to return a number from the normal distribution with mean *mu* and standard deviation *sigma*, use **random.normalvariate(mu, sigma)**:

```
random.normalvariate(100, 15)
```

```
113.62214497887028
```

```
random.normalvariate(100, 15)
```

```
102.40507603179206
```

# Random Selections and Permutations

- ▶ **random.choice()** selects an item at random from a sequence such as a list:

```
list1 = [10, 5, 2, -3.4, "hello"]
random.choice(list1)
```

10

```
random.choice(list1)
```

5

- ▶ **random.shuffle()** randomly shuffles (permutes) the items of the sequence *in place*:

```
random.shuffle(list1)
list1
```

['hello', -3.4, 2, 10, 5]

- ▶ Because the random permutation is made *in place*, the sequence must be mutable: you can't, for example, shuffle tuples

# Random Sequences

- ▶ **random.sample(*population*, *k*)** draws a list of *k* unique elements from a sequence or set (without replacement)

```
lottery_numbers = range(1, 38)
winners = random.sample(lottery_numbers, 6)
winners
[35, 27, 15, 29, 18, 1]
```

## Exercise (29)

- ▶ Define a function **make\_deck()** that creates a randomly shuffled deck of cards
- ▶ The deck contains 52 cards, 13 of each suit: clubs, diamonds, hearts, and spades
- ▶ Each card should be represented as a two-letter string:
  - ▶ The first letter indicates the card suite: ['S', 'H', 'D', 'C']
  - ▶ The second letter indicates the card rank: ['A', '2', '3', '4', '5', '6', '7', '8', '9', 'T', 'J', 'Q', 'K']
  - ▶ e.g., king of spades is "SK", 5 of clubs is "C5", 10 of hearts is "HT", etc.
- ▶ Define another function **deal\_hand(*n*, *deck*)** that draws a hand of *n* random cards from *deck*
- ▶ Sample run:

```
deck = make_deck()
hand = deal_hand(5, deck)
print(hand)

['CK', 'S3', 'HJ', 'D3', 'SK']
```

## Exercise (29) Cont.

- If time permits: define a function `draw_cards(cards)` that display the cards visually using their Unicode representations:

|         | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| U+1F0Ax | 🂱 | 🂲 | 🂳 | 🂴 | 🂵 | 🂶 | 🂷 | 🂸 | 🂹 | 🂺 | 🂻 | 🂼 | 🂽 | 🂾 | 🂿 |   |
| U+1F0Bx |   | 🂴 | 🂵 | 🂶 | 🂷 | 🂸 | 🂹 | 🂺 | 🂻 | 🂽 | 🂻 | 🂼 | 🂽 | 🂾 | 🂿 | 🂿 |
| U+1F0Cx | 🂴 | 🂵 | 🂶 | 🂷 | 🂸 | 🂹 | 🂺 | 🂻 | 🂽 | 🂻 | 🂼 | 🂽 | 🂾 | 🂿 | 🂿 | 🂿 |
| U+1F0Dx | 🂴 | 🂵 | 🂶 | 🂷 | 🂸 | 🂹 | 🂺 | 🂻 | 🂽 | 🂻 | 🂼 | 🂽 | 🂾 | 🂿 | 🂿 | 🂿 |
| U+1F0Ex | 🂱 | 🂲 | 🂳 | 🂴 | 🂵 | 🂶 | 🂷 | 🂸 | 🂹 | 🂺 | 🂻 | 🂼 | 🂽 | 🂾 | 🂿 | 🂿 |
| U+1F0Fx | 🂱 | 🂲 | 🂳 | 🂴 | 🂵 | 🂶 |   |   |   |   |   |   |   |   |   |   |

- Sample usage:

```
deck = make_deck()
hand = deal_hand(5, deck)
print(hand)
draw_cards(hand)
```

```
['S5', 'C5', 'D2', 'H8', 'D9']
🂱 🂲 🂳 🂴 🂵
```

## Exercise (30)

- ▶ The *Monty Hall problem* is a famous conundrum in probability which takes the form of a hypothetical game show
- ▶ The contestant is presented with three doors: behind one is a car and behind each of the other two is a goat
- ▶ The contestant picks a door and then the game show host opens a different door to reveal a goat
- ▶ The host knows which door conceals the car
- ▶ The contestant is then invited to switch to the other closed door or stick with his or her initial choice
- ▶ Write a program in Python that finds the best strategy for winning the car and its probability of winning, by running simulations of the game



# The datetime Module

- ▶ Python's **datetime** module provides classes for manipulating dates and times:
  - ▶ `datetime.date` – an idealized naïve date (unaware of time zones)
  - ▶ `datetime.time` – an idealized time, independent of any particular day
  - ▶ `datetime.datetime` – a combination of a date and time
  - ▶ `datetime.timedelta` – a duration expressing the difference between two date, time, or `datetime` instances to microsecond resolution
  - ▶ `datetime.timezone` – a time zone information object
- ▶ Objects of these types are immutable



# Date Objects

- To create a date object, pass valid year, month and day numbers explicitly or call the `date.today()` constructor:

```
from datetime import date
```

```
my_date = date(2012, 7, 27)
my_date
```

```
datetime.date(2012, 7, 27)
```

```
today = date.today()
today
```

```
datetime.date(2018, 7, 11)
```

```
date(2018, 6, 31) # illegal date
```

```

ValueError Traceback (most recent call last)
<ipython-input-28-fd44869ace8f> in <module>()
----> 1 date(2018, 6, 31)
```

```
ValueError: day is out of range for month
```

- Parsing dates to and from strings is also supported (explained later)

# Date Objects

- ▶ Some useful date attributes and methods:

```
my_date.day
```

27

```
my_date.month
```

7

```
my_date.year
```

2012

```
my_date.weekday() # Monday = 0, Tuesday = 1, ..., Sunday = 6
```

4

```
my_date.ctime() # C-standard time output
```

'Fri Jul 27 00:00:00 2012'

# Date Objects

- ▶ Dates can also be compared (chronologically):

```
my_date < today
```

True

```
my_date == today
```

False

- ▶ Subtracting two dates returns a **datetime.timedelta** object, which represents the difference between the two dates:

```
today - my_date
```

```
datetime.timedelta(2175)
```

- ▶ **timedelta** objects have various attributes for getting the time difference in various units (e.g., days, hours, minutes, etc.)

# Time Objects

- ▶ A **datetime.time** object represents a (local) time of day to the nearest microsecond
- ▶ To create a time object, pass the number of hours, minutes, seconds and microseconds (missing values default to zero)

```
from datetime import time
```

```
lunchtime = time(12, 30)
lunchtime
```

```
datetime.time(12, 30)
```

```
lunchtime.isoformat() # HH:MM:SS
```

```
'12:30:00'
```

```
precise_time = time(4, 46, 36, 501982)
precise_time.isoformat()
```

```
'04:46:36.501982'
```

# DateTime Objects

- ▶ A **datetime.datetime** object contains the information from both the date and time objects: year, month, day, hour, minute, second, microsecond
- ▶ You can pass values for these quantities directly to the `datetime()` constructor, or use the method `now()` to get the current date and time:

```
from datetime import datetime
```

```
now = datetime.now()
now
```

```
datetime.datetime(2018, 7, 29, 6, 2, 4, 400532)
```

```
now.isoformat()
```

```
'2018-07-29T06:02:04.400532'
```

```
now.ctime()
```

```
'Sun Jul 29 06:02:04 2018'
```

# Date and Time Formatting

- date, time and datetime objects support a method **strftime()** to output their values as a string formatted according to the format specifiers listed in following table:

| Specifier | Description                                                       |
|-----------|-------------------------------------------------------------------|
| %a        | Abbreviated weekday (Sun, Mon, etc.)                              |
| %A        | Full weekday (Sunday, Monday, etc.)                               |
| %w        | Weekday number (0=Sunday, 1=Monday, ..., 6=Saturday)              |
| %d        | Zero-padded day of month: 01, 02, 03, ..., 31                     |
| %b        | Abbreviated month name (Jan, Feb, etc.)                           |
| %B        | Full month name (January, February, etc.)                         |
| %m        | Zero-padded month number: 01, 02, ..., 12                         |
| %y        | Year without century (two-digit, zero-padded): 01, 02, ..., 99    |
| %Y        | Year with century (four-digit, zero-padded): 0001, 0002, ... 9999 |

| Specifier | Description                                                        |
|-----------|--------------------------------------------------------------------|
| %H        | 24-hour clock hour, zero-padded: 00, 01, ..., 23                   |
| %I        | 12-hour clock hour, zero-padded: 00, 01, ..., 12                   |
| %p        | AM or PM (or locale equivalent)                                    |
| %M        | Minutes (two-digit, zero-padded): 00, 01, ..., 59                  |
| %S        | Seconds (two-digit, zero-padded): 00, 01, ..., 59                  |
| %f        | Microseconds (six-digit, zero-padded): 000000, 000001, ..., 999999 |
| %%        | The literal % sign                                                 |

# Date and Time Formatting

- ▶ For example:

```
now.strftime("%A, %d %B %Y")
```

```
'Sunday, 29 July 2018'
```

```
now.strftime("%H:%M:%S on %d/%m/%y")
```

```
'06:03:18 on 29/07/18'
```

- ▶ The reverse process, parsing a string into a datetime object is done by **strptime()**:

```
queen_birthtime = datetime.strptime("April 21, 1926 02:40", "%B %d, %Y %H:%M")
print(queen_birthtime.strftime("%I:%M %p on %A, %d %b %Y"))
```

```
02:40 AM on Wednesday, 21 Apr 1926
```

# Adding Dates and Times

- ▶ Using the **datetime** and **timedelta** objects, you can perform date and time addition/subtraction

```
from datetime import datetime
from datetime import timedelta
```

```
Add 1 day
print(datetime.now() + timedelta(days=1))
```

2018-08-06 08:25:40.147347

```
Subtract 60 seconds
print(datetime.now() - timedelta(seconds=60))
```

2018-08-05 08:24:59.764127

```
Pass multiple parameters(1 day and 5 minutes)
print(datetime.now() + timedelta(days=1, minutes=5))
```

2018-08-06 08:31:48.009242

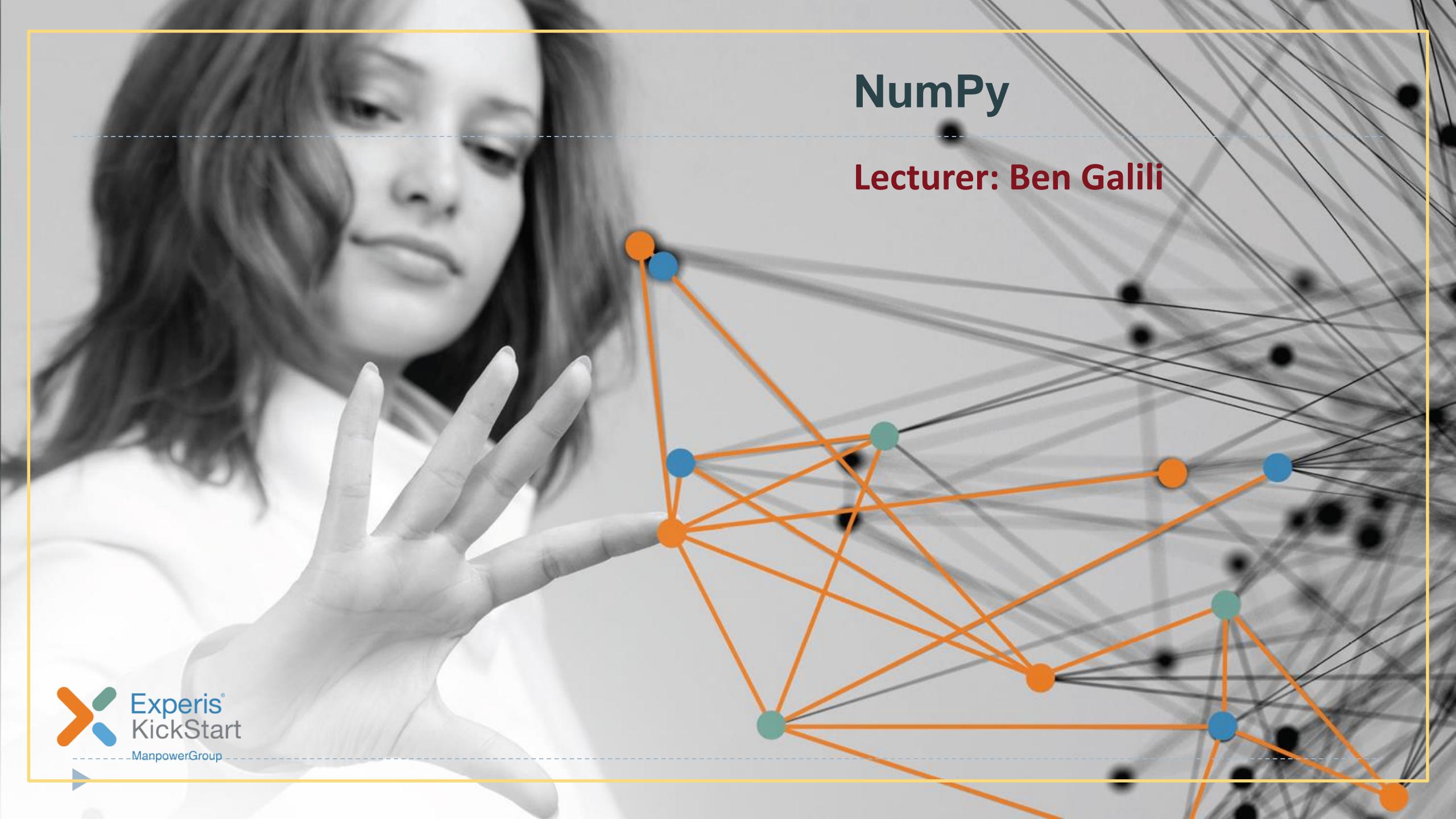
- ▶ Other parameters that can be passed to timedelta: weeks, days, hours, minutes, seconds, microseconds, milliseconds

## Exercise (31)

- ▶ Define a function `get_days_until_birthday(day, month)` that returns the number of days from today until your next birthday as specified by the *day* and *month* arguments
- ▶ Sample run:

```
days = get_days_until_birthday(27, 7)
print("{} days are left until your birthday!".format(days))

356 days are left until your birthday!
```

A black and white photograph of a young woman with long hair, wearing a white shirt. She is reaching out with her right hand towards a network graph. The graph consists of several nodes (dots) connected by lines (edges). Some edges are orange, while others are grey. The background is a light grey with some dark, blurred shapes.

# NumPy

Lecturer: Ben Galili

# NumPy

- ▶ NumPy is the fundamental package for scientific computing with Python
- ▶ Designed to efficiently manipulate multi-dimensional arrays of a single data type
- ▶ Contains useful linear algebra and statistical analysis methods
- ▶ The NumPy implementations of the mathematical operations and algorithms have two main advantages over the “core” Python objects:
  - ▶ They are implemented as precompiled and optimized C code and so are much faster than the core Python alternative
  - ▶ NumPy supports *vectorization*: a single operation can be carried out on an entire array, rather than requiring an explicit loop over the array’s elements
    - ▶ The absence of explicit looping and indexing makes the code cleaner, less error-prone and closer to the standard mathematical notation it reflects

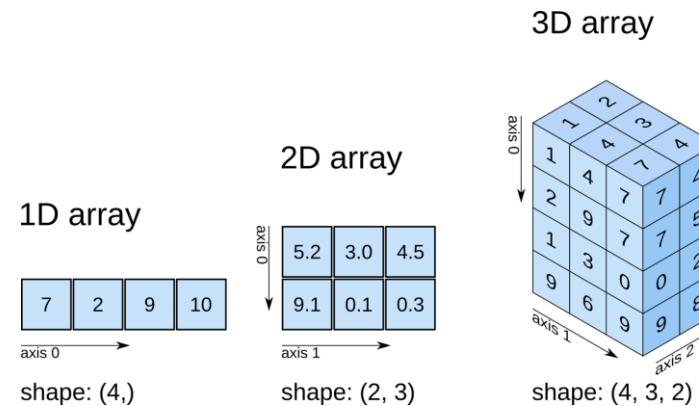


# The NumPy Package

- ▶ All of NumPy's functionality is provided by the **numpy** package
- ▶ To use it, import it by typing **import numpy as np** and then refer to its attributes with the prefix np. (e.g., np.array)

# NumPy Arrays

- ▶ NumPy's main object is the homogeneous multidimensional array of class **ndarray**
- ▶ It is a multidimensional table of elements, indexed by a tuple of positive integers
- ▶ Each element in a NumPy array has the same type, which is specified by an associated *data type* object (**dtype**)
- ▶ The dimensions of a NumPy array are called **axes**
  - ▶ The number of axes an array has is called its **rank**
- ▶ The **shape** of the array defines the size of the array along each of its axes, returned as a tuple of integers



# Basic Array Creation

- ▶ There are several ways to create arrays
- ▶ The simplest way is to call the **np.array()** constructor with a list or tuple of values:

```
import numpy as np

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

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

- ▶ Passing a list of lists creates a two-dimensional array:

```
b = np.array([[0.1, 0.2], [0.3, 0.4]])

b
array([[0.1, 0.2],
 [0.3, 0.4]])
```

# Basic Array Creation

- ▶ The data type is deduced from the type of the elements in the sequence and “upcast” to the most general type if they are of mixed but compatible types:

```
np.array([-1, 0, 0.5]) # mixture of int and float: upcast to float
array([-1. , 0. , 0.5])
```

- ▶ You can also explicitly set the data type using the optional **dtype** argument:

```
np.array([0, 4, -4], dtype=complex)
array([0.+0.j, 4.+0.j, -4.+0.j])
```

# Basic Array Creation

- ▶ If your array is large or you don't know the element values at the time of creation, there are several methods to declare an array of a particular shape filled with default or arbitrary values
- ▶ The simplest and fastest, **np.empty()**, takes a tuple of the array's shape and creates the array without initializing its elements
  - ▶ The initial element values are undefined (typically random junk)

```
np.empty((3, 3))
```

```
array([[0.0000000e+000, 0.0000000e+000, 0.0000000e+000],
 [0.0000000e+000, 0.0000000e+000, 6.81810591e-321],
 [6.28110754e-312, 0.0000000e+000, 3.40724714e-085]])
```

- ▶ There are also helper methods **np.zeros()** and **np.ones()**, which create an array of the specified shape with elements prefilled with 0 and 1 respectively
- ▶ **np.empty()**, **np.zeros()** and **np.ones()** also take the optional **dtype** argument

# Basic Array Creation

## ► Examples:

```
np.zeros((3, 2)) # default dtype is 'float'
```

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

```
np.ones((3, 3), dtype=int)
```

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

- If you already have an array and would like to create another with the same shape, **np.empty\_like()**, **np.zeros\_like()** and **np.ones\_like()** will do that for you:

```
a
```

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

```
np.ones_like(a)
```

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

# Initializing an Array from a Sequence

- ▶ To create an array containing a sequence of numbers there are two methods:  
**np.arange()** and **np.linspace()**
- ▶ **np.arange(start, stop, step)** is the NumPy equivalent of range(), except that it can generate floating point sequences
- ▶ It also actually allocates the memory for the elements in an ndarray instead of returning a generator-like object

```
np.arange(7)
```

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

```
np.arange(1.5, 3.0, 0.5)
```

```
array([1.5, 2. , 2.5])
```

- ▶ As with range(), the generated arrays don't include the last elements

# Initializing an Array from a Sequence

- ▶ **np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)**  
generates an evenly spaced array over the specified interval
  - ▶ start – the starting value of the sequence
  - ▶ stop – the end value of the sequence (the interval includes this value)
  - ▶ num – number of samples to generate (default is 50)
  - ▶ endpoint – If True, *stop* is the last sample. Otherwise, it is not included
  - ▶ retstep – If True, return (*samples, step*), where *step* is the spacing between samples
  - ▶ dtype – The type of the output array (default is float)
- ▶ For example, to generate an evenly spaced array of five numbers between 1 and 20:

```
np.linspace(1, 20, 5)
```

```
array([1. , 5.75, 10.5 , 15.25, 20.])
```

# Initializing an Array from a Sequence

- ▶ Setting **retstep** to True returns the number spacing (step size):

```
x, dx = np.linspace(0, 2 * np.pi, 100, retstep=True)
```

```
dx
```

```
0.06346651825433926
```

- ▶ This saves you from calculating  $dx = (\text{end-start})/(\text{num}-1)$  separately
- ▶ In this example 100 points between 0 and  $2\pi$  inclusive are spaced by  $2\pi/99=0.063466\dots$
- ▶ Setting **endpoint** to False omits the final point in the sequence, as for np.arange:

```
np.linspace(1, 5, 5, endpoint=False)
```

```
array([1. , 1.8, 2.6, 3.4, 4.2])
```

- ▶ The step size in this case is  $dx = (\text{end-start})/\text{num}$

## Exercise (1)

- ▶ Create a  $3 \times 3$  matrix with values ranging from 0 to 8
- ▶ Create an array with values ranging from 10 to 49
- ▶ Create a vector of size 10 containing only zeros of integer type
- ▶ Create a Hilbert matrix of size  $N \times N$  ( $N$  is given by the user)

- ▶ Hilbert matrix is a square matrix with entries being the unit fractions

$$H_{ij} = \frac{1}{i+j-1}$$

- ▶ For example, this is the  $5 \times 5$  Hilbert matrix:

$$\begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{pmatrix}$$

# ndarray Attributes

- ▶ The following table shows some useful attributes of a NumPy array:

| Attribute | Description                                                                                         |
|-----------|-----------------------------------------------------------------------------------------------------|
| shape     | The array dimensions: the size of the array along each of its axes, returned as a tuple of integers |
| ndim      | Number of axes (dimensions)<br>Note that <code>ndim == len(shape)</code>                            |
| size      | The total number of elements in the array, equal to the product of the elements of shape            |
| dtype     | The array's data type                                                                               |
| itemsize  | The size in bytes of each element                                                                   |
| data      | The “buffer” in memory containing the actual elements of the array                                  |

# ndarray Attributes

```
a = np.array([[1, 0, 1], [0, 1, 0]])
a.shape # 2 rows, 3 columns
```

```
(2, 3)
```

```
a.ndim # rank (number of dimensions)
```

```
2
```

```
a.size # total number of elements
```

```
6
```

```
a.dtype
```

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

```
a.itemsize # each item consumes 4 bytes
```

```
4
```

```
a.data
```

```
<memory at 0x0000015E57005048>
```

# NumPy Basic Data Types (dtypes)

- ▶ The NumPy arrays we've created so far contained either integers or floating point numbers, and we've let Python take care of the details of how these are represented
- ▶ NumPy provides ways of specifying the elements type using *data type* objects
- ▶ This is useful for scenarios such as:
  - ▶ When interfacing with the underlying compiled C code, the elements of a NumPy array must be stored in a compatible format
  - ▶ Reading arrays from a binary file generated by a different computer
- ▶ For example, an unsigned integer (C-type `uint16_t`) is stored in 2 bytes of memory:
  - ▶ Such a number can take a value between 0 and  $2^{16} - 1 = 65535$
  - ▶ No equivalent native Python type exists for this exact representation
    - ▶ Python integers are signed and memory is dynamically assigned for them by their size
  - ▶ NumPy defines a data type object `np.uint16` to describe data stored in this way

# Common NumPy Data Types

| Data Type | Description                                                                                                    |
|-----------|----------------------------------------------------------------------------------------------------------------|
| int_      | The default integer type, corresponding to C's long (platform-dependent)                                       |
| int8      | Integer in a single byte: -128 to 127                                                                          |
| int16     | Integer in 2 bytes: -32768 to 32767                                                                            |
| int32     | Integer in 4 bytes: -2147483648 to 2147483647                                                                  |
| int64     | Integer in 8 bytes: $-2^{63}$ to $2^{63} - 1$                                                                  |
| uint8     | Unsigned integer in a single byte: 0 to 255                                                                    |
| uint16    | Unsigned integer in 2 bytes: 0 to 65535                                                                        |
| uint32    | Unsigned integer in 4 bytes: 0 to 4294967295                                                                   |
| uint64    | Unsigned integer in 8 bytes: 0 to $2^{64} - 1$                                                                 |
| float_    | The default floating point number type, another name for float64                                               |
| float32   | Single-precision, signed float: $\sim 10^{-38}$ to $\sim 10^{38}$ with $\sim 7$ decimal digits of precision    |
| float64   | Double-precision, signed float: $\sim 10^{-308}$ to $\sim 10^{308}$ with $\sim 15$ decimal digits of precision |
| bool_     | The default boolean type represented by a single byte                                                          |
| complex_  | The default complex number type (Double-precision floating point real and imaginary components)                |

# NumPy Basic Data Types (dtypes)

- ▶ All data types exist within the numpy package, e.g., np.uint16
- ▶ The data types that get created by default when using the native Python numerical types are those with a trailing underscore: np.float\_, np.complex\_ and np.bool\_
- ▶ To create a NumPy array of values using a particular data type, use the **dtype** argument of any array constructor function (such as np.array, np.zeros, etc.)
- ▶ The following example creates a  $3 \times 3$  array of unsigned, 8-bit (1-byte) integers:

```
np.ones((3, 3), dtype=np.uint8)

array([[1, 1, 1],
 [1, 1, 1],
 [1, 1, 1]], dtype=uint8)
```

- ▶ It is common to specify the **dtype** using a string consisting of a letter indicating the broad category of data type (integer, unsigned integer, complex number, etc.), optionally followed by the byte size of the type

# Indexing

- ▶ An array is indexed by a tuple of integers
- ▶ This is a little different from indexing a conventional Python list of lists: instead of  $a[i][j]$ , we refer to the index of the required element as a tuple of integers,  $a[i, j]$ :

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

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

```
a[0, 1] # same as a[(0, 1)]
```

```
2
```

```
a[1, 1] = 0
a
```

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

```
a[0] # get the first row
```

```
array([1, 2])
```

# Indexing

- ▶ As for Python sequences negative indexes count from the end of the axis:

```
a[0, -1] = 8
a
```

```
array([[1, 8],
 [3, 0]])
```

- ▶ Trying to access an element outside the array bounds will result in an IndexError:

```
a[0, 2]
```

```

IndexError Traceback (most recent call last)
<ipython-input-7-67f04df8cd08> in <module>()
----> 1 a[0, 2]

IndexError: index 2 is out of bounds for axis 1 with size 2
```

- ▶ More advanced techniques for indexing will be discussed later

## Exercise (2)

- ▶ A *magic square* is an  $N \times N$  grid of numbers in which the entries in each row, column and main diagonal sum to the same number (equal to  $N(N^2 + 1)/2$ ).
- ▶ A method for constructing a magic square for odd  $N$  is as follows:
  1. Start in the middle of the top row, and let  $n = 1$ .
  2. Insert  $n$  into the current grid position.
  3. If  $n = N^2$  the grid is complete so stop. Otherwise, increment  $n$ .
  4. Move diagonally up and right, wrapping to the first column or last row if the move leads outside the grid. If this cell is already filled, move vertically down one space **instead**.
  5. Return to step 2.
- ▶ Write a program that gets  $N$  from the user, creates and displays a magic square of size  $N \times N$ .

# Universal Functions (ufuncs)

- ▶ A universal function (**ufunc**) is a function that operates on ndarrays in an element-by-element fashion, producing an array in return without the need for an explicit loop
  - ▶ For example, the function **np.sqrt(a)** returns the square root of an array, element-wise
- ▶ There are currently more than 60 universal functions
  - ▶ Including many of the familiar mathematical functions from the math module
  - ▶ Many of them are implemented in compiled C code
  - ▶ The full list can be found at <https://docs.scipy.org/doc/numpy-1.14.0/reference/ufuncs.html>
- ▶ Some of the ufuncs are called automatically when the relevant infix notation is used
  - ▶ e.g., **np.add(a, b)** is called internally when **a + b** is written, and a or b is an ndarray
- ▶ You may still want to call the ufunc directly in order to specify optional arguments
- ▶ Universal functions are the way NumPy allows for *vectorization*, which promotes clean, efficient and easy-to-maintain code

# Universal Functions (ufuncs)

```
x = np.linspace(1, 5, 5)
x
```

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

```
x**2
```

```
array([1., 4., 9., 16., 25.])
```

```
x - 1
```

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

```
np.sqrt(x - 1)
```

```
array([0. , 1. , 1.41421356, 1.73205081, 2.])
```

```
y = np.exp(-np.linspace(0, 2, 5)) # y = [e^0, e^(-0.5), ..., e^(-2)]
y
```

```
array([1. , 0.60653066, 0.36787944, 0.22313016, 0.13533528])
```

```
np.sin(x - y)
```

```
array([0. , 0.98431873, 0.48771645, -0.59340065, -0.98842844])
```

# Universal Functions (ufuncs)

- ▶ Array multiplication occurs **elementwise**

```
a = np.array ([[1, 2], [3, 4]])
b = np.array ([[1, 0], [0, 1]])
a * b # elementwise multiplication
```

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

- ▶ Matrix multiplication is implemented by NumPy's dot function:

```
a.dot(b) # or np.dot(a, b)
```

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

# Universal Functions (ufuncs)

- ▶ Comparison and logic operators are also vectorized and result in arrays of booleans
- ▶ The boolean operations *not*, *and*, *or* are implemented on boolean arrays with the operators `~`, `&` and `|` respectively

```
a = np.linspace(0, 100, 5)
print(a)
```

```
[0. 25. 50. 75. 100.]
```

```
print(a > 50)
```

```
[False False False True True]
```

```
print((a < 10) | (a > 50))
```

```
[True False False True True]
```

- ▶ In the last example the parentheses around the comparisons are necessary, since bitwise operators (`~`, `&`, and `|`) have higher precedence than comparison operators

# NumPy's Special Values

- ▶ NumPy defines two special values to represent the outcome of calculations, which are not mathematically defined or not finite
- ▶ **np.nan** (“not a number,” NaN) represents the outcome of a calculation that is not a well-defined mathematical operation (e.g.,  $0/0$ )
- ▶ **np.inf** represents infinity

```
a = np.arange(4)
a = a / 0 # [0/0 1/0 2/0 3/0]
a
```

```
C:\Users\roi\Anaconda3\lib\site-packages\ipykernel_launcher.py:
2: RuntimeWarning: divide by zero encountered in true_divide

C:\Users\roi\Anaconda3\lib\site-packages\ipykernel_launcher.py:
2: RuntimeWarning: invalid value encountered in true_divide
```

```
array([nan, inf, inf, inf])
```

# Changing the Shape of the Array

- ▶ Whatever the rank of an array, its elements are stored in sequential memory locations that are addressed by a single
  - ▶ Knowing the shape of the array, Python is able to resolve a tuple of indexes into a single
- ▶ NumPy's arrays are stored in memory in C-style, *row-major* order
  - ▶ i.e., the elements of the last (rightmost) index are stored contiguously
- ▶ For example, the array

```
import numpy as np

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

```
[[1 2]
 [3 4]]
```

- ▶ is stored in memory as the sequential elements [1,2,3,4]

# Flattening an Array

- ▶ NumPy provides two methods to flatten a multidimensional array into one dimension:
  - ▶ **fatten()** – return an independent copy of the elements
  - ▶ **ravel()** – returns a view of the flattened array, i.e., a new one-dimensional array that references the elements of the original array
- ▶ Example for using flatten():

```
a = np.array([[1,2,3], [4,5,6], [7,8,9]])
b = a.flatten() # create an independent, flattened copy of a
b
```

```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
b[3] = 0
b
```

```
array([1, 2, 3, 0, 5, 6, 7, 8, 9])
```

```
a # a is unchanged
```

```
array([[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]])
```

# Flattening an Array

- In contrast, the flattened array created by taking a view with **ravel()** refers to the same underlying data:

```
c = a.ravel()
c
```

```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
c[3] = 0
c
```

```
array([1, 2, 3, 0, 5, 6, 7, 8, 9])
```

```
a
```

```
array([[1, 2, 3],
 [0, 5, 6],
 [7, 8, 9]])
```

# Resizing an Array

- ▶ An array may be resized (in place) to a compatible shape with the `resize()` method, which takes the new dimensions as its arguments
  - ▶ A comptiable shape is a shape with the same total number of elements
- ▶ Changing the size of the array is only allowed if the array doesn't reference another array's data and doesn't have references to it, in which case:
  - ▶ Resizing to a larger shape pads with zeros
  - ▶ Resizing to a smaller shape truncates the array
- ▶ The purpose of the reference count check is to make sure you don't use this array as a buffer for another Python object and then reallocate the memory

# Resizing an Array

## ► Examples:

```
a = np.array([1, 2, 3, 4])
a.resize((2, 2)) # reshapes a in place, doesn't return anything
print(a)
```

```
[[1 2]
 [3 4]]
```

```
b = np.array([1, 2, 3, 4])
b.resize(3, 2) # OK: nothing else references b
print(b)
```

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

```
c = np.array([1, 2, 3, 4])
c.resize(1, 2) # OK: nothing else references c
print(c)
```

```
[[1 2]]
```

# Resizing an Array

- ▶ Referencing an array prevents resizing:

```
a = np.array([1, 2, 3, 4])
d = a
a.resize(1, 1)
```

```

ValueError Traceback (most recent call last)
<ipython-input-12-a11d7155882e> in <module>()
 1 a = np.array([1, 2, 3, 4])
 2 d = a
----> 3 a.resize(1, 1)
```

**ValueError:** cannot resize an array that references or is referenced by another array in this way. Use the resize function

- ▶ Unless *refcheck* is False:

```
a = np.array([1, 2, 3, 4])
d = a
a.resize(1, 1, refcheck=False)
print(a)
```

```
[[1]]
```

# Reshaping an Array

- ▶ `reshape()` returns a view on an array with its elements reshaped as required
  - ▶ The original array is not modified
  - ▶ Thus, `reshape()` cannot change the size of the array

```
a = np.arange(6)
b = a.reshape(2, 3)
b
```

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

```
b.resize(3, 2) # OK: same number of elements
```

```
b.resize(2, 2) # Not OK: b is a view (shares the same data) as a
```

```

ValueError Traceback (most recent call
1 last)
<ipython-input-33-3911e7fb01b8> in <module>()
----> 1 b.resize(2, 2) # Not OK: b is a view (shares the same data)
 as a
```

```
ValueError: cannot resize this array: it does not own its data
```

# Transposing an Array

- ▶ The method **transpose()** returns a view of an array with the dimensions reversed
- ▶ For a two-dimensional array, this is the usual matrix transpose:

```
a = np.arange(6).reshape(3, 2)
a
```

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

```
a.transpose() # or simply a.T
```

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

- ▶ Transposing a one-dimensional array returns the array unchanged:

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

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

# Merging Arrays

- ▶ NumPy provides several methods to merge arrays in different ways:
  - ▶ **np.vstack()** – stack arrays vertically (in sequential rows)
  - ▶ **np.hstack()** – stack arrays horizontally (in sequential columns)
- ▶ The arrays created contain an independent copy of the data from the original arrays

```
a = np.array([0, 0, 0, 0])
b = np.array([1, 1, 1, 1])
c = np.array([2, 2, 2, 2])
np.vstack((a, b, c))
```

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

```
np.hstack((a, b, c))
```

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

# Splitting Arrays

- ▶ The inverse operations, `np.vsplit()`, `np.hsplit()` and `np.dsplit()` split a single array into multiple arrays by rows, columns or depth
- ▶ The arrays returned are *views* on the original data
- ▶ In addition to the array to be split, these methods require an argument indicating how to split the array
- ▶ If this argument is a single integer, the array is split into that number of equal-sized arrays along the appropriate axis:

```
a = np.arange(6)
a
```

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

```
np.hsplit(a, 3)
```

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

# Splitting Arrays

- If the second argument is a sequence of integer indexes, the array is split on those indexes:

```
a
```

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

```
np.hsplit(a, (2, 3, 5))
```

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

## Exercise

- ▶ Create a  $3 \times 3$  array of ones
- ▶ Add the row (2, 2, 2) to the array
- ▶ Add the column (3, 3, 3, 3) to the array
- ▶ The result should be:  
[[1 1 1 3]  
 [1 1 1 3]  
 [1 1 1 3]  
 [2 2 2 3]]

# Slicing

- ▶ Slicing and striding is supported in the same way as in Python sequences
- ▶ For one-dimensional arrays there is only one index:

```
a = np.arange(1, 7)
a
```

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

```
a[1:4]
```

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

```
a[1:4:2]
```

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

```
a[5::-2]
```

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

# Slicing

- Multidimensional arrays have an index for each axis
  - If you want to select every item along a particular axis, replace its index with a single colon

```
a = np.arange(1, 13).reshape(4, 3)
a
```

```
array([[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9],
 [10, 11, 12]])
```

```
a[0:2, 0:2] # the first two rows and two columns
```

```
array([[1, 2],
 [4, 5]])
```

```
a[2, :] # everything in the third row
```

```
array([7, 8, 9])
```

```
a[:, 1] # everything in the second column
```

```
array([2, 5, 8, 11])
```

```
a[1:-1, 1:] # middle rows, second column onwards
```

```
array([[5, 6],
 [8, 9]])
```

|    |    |    |
|----|----|----|
| 1  | 2  | 3  |
| 4  | 5  | 6  |
| 7  | 8  | 9  |
| 10 | 11 | 12 |

a[2, :]  
(a)

|    |    |    |
|----|----|----|
| 1  | 2  | 3  |
| 4  | 5  | 6  |
| 7  | 8  | 9  |
| 10 | 11 | 12 |

a[:, 1]  
(b)

|    |    |    |
|----|----|----|
| 1  | 2  | 3  |
| 4  | 5  | 6  |
| 7  | 8  | 9  |
| 10 | 11 | 12 |

a[1:-1, 1:]  
(c)

|    |    |    |
|----|----|----|
| 1  | 2  | 3  |
| 4  | 5  | 6  |
| 7  | 8  | 9  |
| 10 | 11 | 12 |

a[2, ::2]  
(d)

|    |    |    |
|----|----|----|
| 1  | 2  | 3  |
| 4  | 5  | 6  |
| 7  | 8  | 9  |
| 10 | 11 | 12 |

a[2::, ::2]  
(e)

|    |    |    |
|----|----|----|
| 1  | 2  | 3  |
| 4  | 5  | 6  |
| 7  | 8  | 9  |
| 10 | 11 | 12 |

a[1::2, ::2]  
(f)

# Slicing

- ▶ The colon and ellipsis syntax also works for assignment:

```
a = np.arange(1, 13).reshape(4, 3)
a
```

```
array([[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9],
 [10, 11, 12]])
```

```
a[:, 1] = 0 # set all elements in the second column to 0
a
```

```
array([[1, 0, 3],
 [4, 0, 6],
 [7, 0, 9],
 [10, 0, 12]])
```

# Exercise

- ▶ A  $3 \times 4 \times 4$  array is created with:

```
a = np.arange(1, 49).reshape(3, 4, 4)
```

- ▶ Slice this array to obtain the following:

- ▶ The  $1 \times 4$  array: [ 9 10 11 12]

- ▶ The  $4 \times 4$  array: [[33 34 35 36]  
[37 38 39 40]  
[41 42 43 44]  
[45 46 47 48]]

- ▶ The  $3 \times 2$  array: [[ 5 6]  
[21 22]  
[37 38]]

- ▶ The  $4 \times 2$  array: [[36 35]  
[40 39]  
[44 43]  
[48 47]]

- ▶ The  $3 \times 4$  array: [[13 9 5 1]  
[29 25 21 17]  
[45 41 37 33]]

# Boolean Indexing

- ▶ Instead of indexing an array with a sequence of integers, it is also possible to use an array of boolean values
  - ▶ The True elements of this indexing array identify elements in the target array to be returned

```
a = np.array([-2, -1, 0, 1, 2])
idx = np.array([False, True, False, True, True])
a[idx]
```

```
array([-1, 1, 2])
```

- ▶ Because comparisons are vectorized across arrays just like mathematical operations, this leads to some useful shortcuts:

```
a[a < 0] = 0 # set all negative elements to zero
print(a)
```

```
[0 0 0 1 2]
```

## Exercise

- ▶ You are given an array containing a list of years:

```
years = np.array([1900, 1904, 1990, 1993, 2000, 2014, 2016, 2100])
```

- ▶ Create an array that contains only the leap years from the list
  - ▶ without using Python loops
- ▶ The result should be: [1900 1904 2000 2016 2100]

# Maximum and Minimum

- ▶ The methods **min()** and **max()** return the minimum and maximum values in the array
- ▶ By default, a single value for the flattened array is returned
- ▶ To find maximum and minimum values along a given axis, use the **axis** argument

```
a = np.array([[3, 0, -1, 1], [2, -1, -2, 4], [1, 7, 0, 4]])
print(a)
```

```
[[3 0 -1 1]
 [2 -1 -2 4]
 [1 7 0 4]]
```

```
a.min() # "global" minimum
```

```
-2
```

```
a.max() # "global" maximum
```

```
7
```

```
print(a.max(axis=0)) # maxima in each column
```

```
[3 7 0 4]
```

```
print(a.max(axis=1)) # maxima in each row
```

```
[3 4 7]
```

# Maximum and Minimum

- The methods **argmin()** and **argmax()** return the indexes of the maximum and minimum values in the array
- By default, the index returned is into the *flattened* array
- If more than one equal maximum/minimum exists, the index of the first is returned

```
a.argmin()
```

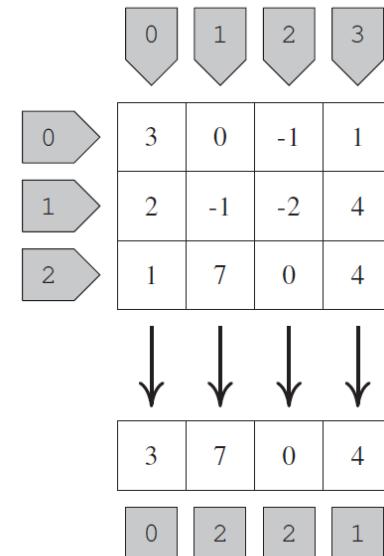
```
6
```

```
print(a.argmax(axis=0))
```

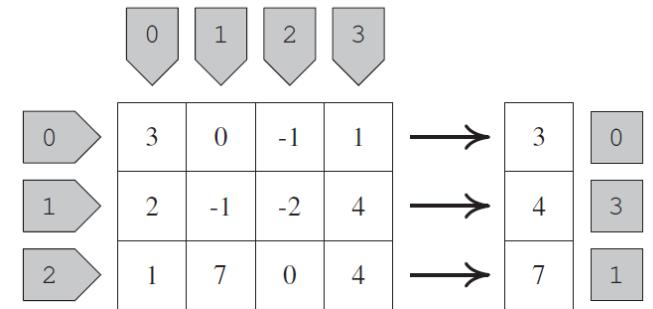
```
[0 2 2 1]
```

```
print(a.argmax(axis=1))
```

```
[0 3 1]
```



(a) axis=0



(b) axis=1

## Exercise

- ▶ Create a function **closest(arr, v)** which takes a NumPy array and a scalar v, and returns the closest value in the array to v
  - ▶ Don't use Python loops!
- ▶ Sample usage of the function:

```
a = np.arange(100)
print(closest(a, 45.7))
```

46

# Sorting an Array

- ▶ NumPy arrays can be sorted in place with the **sort()** method
- ▶ By default, this method sorts multidimensional arrays along their **last axis**
- ▶ To sort along some other axis, set the **axis** argument

```
a = np.array([5, -1, 2, 4, 0, 4])
a.sort()
print(a)
```

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

```
b = np.array([[0, 3, -2], [7, 1, 3], [4, 0, -1]])
print(b)
```

```
[[0 3 -2]
 [7 1 3]
 [4 0 -1]]
```

```
b.sort() # sort the numbers along each row, the same as b.sort(axis=1)
print(b)
```

```
[[-2 0 3]
 [1 3 7]
 [-1 0 4]]
```

# Sorting an Array

- ▶ To sort the numbers in each column set axis=0:

```
b = np.array([[0, 3, -2], [7, 1, 3], [4, 0, -1]])
b.sort(axis=0) # sort the numbers along each column
print(b)
```

```
[[0 0 -2]
 [4 1 -1]
 [7 3 3]]
```

- ▶ The sorting algorithm used is quicksort, which is a good general purpose choice
- ▶ Some arrays can be sorted faster with mergesort or heapsort algorithms
- ▶ These can be selected by setting the optional **kind** argument
  - ▶ e.g., `b.sort(axis=1, kind='heapsort')`

# Sorting an Array

- ▶ **np.argsort()** returns the *indexes* that would sort an array rather than the sorted elements themselves:

```
a = np.array([3, 0, -1, 1])
np.argsort(a)

array([2, 1, 3, 0], dtype=int64)
```

```
a[np.argsort(a)]

array([-1, 0, 1, 3])
```

- ▶ **np.searchsorted()** takes a *sorted* array, a, and one or more values, v, and returns the indexes in a at which the values should be entered to maintain its order:

```
a = np.array([1, 2, 3, 4])
np.searchsorted(a, 3.5)

3
```

## Exercise

- ▶ Create an array of size  $3 \times 3$ , e.g.,  $\begin{bmatrix} [0 & 3 & -2] \\ [7 & 1 & 3] \\ [4 & 0 & -1] \end{bmatrix}$
- ▶ Sort the array by the second column
  - ▶ For the array above the result should be:  $\begin{bmatrix} [4 & 0 & -1] \\ [7 & 1 & 3] \\ [0 & 3 & -2] \end{bmatrix}$

# Logic and Comparisons

- ▶ NumPy provides a set of methods for comparing and performing logical operations on arrays elementwise:

| Function                                     | Description                                                                                                      |
|----------------------------------------------|------------------------------------------------------------------------------------------------------------------|
| <code>np.all(<i>a</i>)</code>                | Determine whether <i>all</i> array elements of <i>a</i> evaluate to True                                         |
| <code>np.any(<i>a</i>)</code>                | Determine whether <i>any</i> array element of <i>a</i> evaluates to True                                         |
| <code>np.isclose(<i>a</i>, <i>b</i>)</code>  | Return a boolean array of the comparison between arrays <i>a</i> and <i>b</i> for equality within some tolerance |
| <code>np.allclose(<i>a</i>, <i>b</i>)</code> | Return True if all the elements in the arrays <i>a</i> and <i>b</i> are equal within some tolerance              |

# Logic and Comparisons

- ▶ np.all and np.any work the same as Python's built-in functions of the same name:

```
a = np.array([[1, 2, 0, 3], [4, 0, 1, 1]])
```

```
np.any(a)
```

True

```
np.all(a) # Some (but not all) elements evaluate to True
```

False

# Logic and Comparisons

- ▶ Because the representation of floating point numbers is not exact, comparing two float or complex arrays with the operator `==` is not reliable and is not recommended
- ▶ Instead, the best we can do is see if two values are “close” to one another within some (typically small) absolute or relative tolerance
- ▶ The function `np.isclose(a, b)` returns True for elements satisfying  $\text{abs}(a-b) \leq (\text{atol} + \text{rtol} * \text{abs}(b))$ 
  - ▶ By default, `atol` (absolute tolerance) =  $10^{-8}$ , `rtol` (relative tolerance) =  $10^{-5}$
  - ▶ Can be changed by setting the corresponding arguments

```
a = np.array([1.66e-27, 1.38e-23, 6.63e-34, 6.02e23])
b = np.array([1.66e-27, 1.66e-27, 1.66e-27, 6.00e23])
np.isclose(a, b)

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

# Structured Arrays

- ▶ **Structured arrays (aka record arrays)** are arrays consisting of rows of values, where each value may have its own data type and name
- ▶ This is very much like a table of data with rows (records) consisting of values that fall into columns (fields)
- ▶ The structure of a record array is defined by its dtype:

```
a = np.zeros(5, dtype='int8, float32, complex')
print(a)

[(0, 0., 0.+0.j) (0, 0., 0.+0.j) (0, 0., 0.+0.j) (0, 0., 0.+0.j)
 (0, 0., 0.+0.j)]
```

```
a.dtype

dtype([('f0', 'i1'), ('f1', '<f4'), ('f2', '<c16')])
```

- ▶ Here we've created an array of five records, each of which has three fields: a single-byte, signed integer (i1), a single-precision floating point number, which is stored as a little-endian 4-byte sequence (<f4), and a complex number which is stored in 16-bytes, little-endian (<c16)

# Structured Arrays

- ▶ Because we did not explicitly name the fields, they are given the default names 'f0', 'f1' and 'f2'
- ▶ To name the fields of our structured array explicitly, pass the dtype constructor a list of (name, dtype descriptor) tuples:

```
dt = np.dtype([('time', 'f8'), ('signal', 'i4')])
a = np.zeros(5, dtype=dt)
a

array([(0., 0), (0., 0), (0., 0), (0., 0), (0., 0)],
 dtype=[('time', '<f8'), ('signal', '<i4')])
```

- ▶ Assigning records in a structured array is as expected:

```
a[0] = (0.5, 4)
a[1:3] = [(0.2, -3), (0.1, -5)]
a

array([(0.5, 4), (0.2, -3), (0.1, -5), (0., 0), (0., 0)],
 dtype=[('time', '<f8'), ('signal', '<i4')])
```

# Structured Arrays

- ▶ The real power of this approach is in the ability to reference a field by its name
- ▶ For example, to set the 'time' column in our array to a linear sequence:

```
a['time'] = np.linspace(0, 2, 5)
print(a)
```

```
[(0. , 4) (0.5, -3) (1. , -5) (1.5, 0) (2. , 0)]
```

```
print(a['time'][-1])
```

```
2.0
```

- ▶ Likewise, to obtain a view on a column, refer to it by name:

```
print(a['time'])
```

```
[0. 0.5 1. 1.5 2.]
```

```
print(a['signal'].min())
```

```
-5
```

## Exercise

- ▶ Turn the following data concerning various species of cetacean into a NumPy structured array and order it by (a) mass and (b) population
- ▶ Determine in each case the index at which *Bryde's whale* (population: 100000, mass: 25 tonnes) should be inserted to keep the array ordered

| Name                        | Population | Mass/tonnes |
|-----------------------------|------------|-------------|
| Bowhead whale               | 9000       | 60          |
| Blue whale                  | 20000      | 120         |
| Fin whale                   | 100000     | 70          |
| Pacific white-sided dolphin | 1000000    | 0.15        |
| Killer whale                | 100000     | 4.5         |
| Sperm whale                 | 2000000    | 50          |
| North Atlantic right whale  | 300        | 75          |
| Southern right whale        | 7000       | 70          |

# Random Sampling

- ▶ NumPy's **random** module provides methods for obtaining random numbers and choosing random entries from an array
- ▶ As with the core library's random module, np.random uses a Mersenne Twister pseudorandom number generator (PRNG)
- ▶ The way it seeds itself is operating-system dependent, but it can be reseeded with any hashable object by calling **np.random.seed()**

# Random Floating Point Numbers

- ▶ **np.random.random\_sample(size)** creates an array of the specified shape filled with numbers sampled randomly from the uniform distribution over [0, 1)
  - ▶ e.g., if the given shape is (m, n) then m \* n samples are drawn
  - ▶ If no argument is specified, a single random number is returned

```
np.random.random_sample((3, 2))
```

```
array([[0.38246199, 0.98323089],
 [0.46676289, 0.85994041],
 [0.68030754, 0.45049925]])
```

- ▶ The **np.random.rand()** method is similar, but is passed the dimensions of the desired array as separate arguments:

```
np.random.rand(3, 2)
```

```
array([[0.01326496, 0.94220176],
 [0.56328822, 0.3854165],
 [0.01596625, 0.23089383]])
```

# Random Floating Point Numbers

- ▶ To sample  $\text{Unif}[a, b]$  multiply the output of `random_sample()` by  $(b-a)$  and add  $a$
- ▶ For example, to sample 10 random numbers between 10 and 20:

```
a, b = 10, 20
a + (b - a) * np.random.rand(3, 2)

array([[12.41025466, 16.83263519],
 [16.09996658, 18.33194912],
 [11.73364654, 13.91060608]])
```

# Random Integers

- ▶ The method **np.random.randint(low, high, size)** takes up to three arguments:
  - ▶ If both low and high are supplied, then the random number(s) are sampled from [low, high)
  - ▶ If low is supplied but high is not, then the sampled interval is [0,low)
  - ▶ size is the shape of the array of random integers desired
    - ▶ If it is omitted, a single random integer is returned

```
np.random.randint(4) # random integer from [0, 4)
```

```
2
```

```
np.random.randint(4, size=10) # 10 random ints from [0, 4)
```

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

```
np.random.randint(1, 10, (3, 5)) # array of random ints in [1, 10)
```

```
array([[8, 7, 9, 8, 5],
 [2, 5, 8, 9, 9],
 [1, 9, 7, 9, 8]])
```

# Random Selections

- ▶ `np.random.choice(a, size=None, replace=True, p=None)` returns a random element a one-dimensional sequence. Its arguments are:
  - ▶ `size` – the output shape
  - ▶ `replace` – whether the sample is with or without replacement
  - ▶ `p` – The probabilities associated with each entry in a
    - ▶ If not given the sample assumes a uniform distribution over all entries in a

```
a = np.array([1, 2, 0, -1, 1])
np.random.choice(a) # a random selection from a
```

0

```
np.random.choice(a, 6) # 6 random selections from a with replacement
array([-1, 0, 0, 1, 0, 1])
```

```
np.random.choice(a, (2, 2), replace=False) # random selections with replacement
array([[1, 2],
 [-1, 1]])
```

# Random Selections

- When sampling without replacement, it is not possible to draw a larger number of elements than there are in the original population:

```
np.random.choice(a, (3, 3), replace=False)
```

```
ValueError: Cannot take a larger sample than population when 'replace=False'
```

- To specify the probability of each element being selected, pass a sequence of the same length as the population to be sampled as the argument p
  - The probabilities should sum to 1

```
a = np.array([1, 2, 0, -1, 1])
np.random.choice(a, 5, p=[0.1, 0.1, 0, 0.8, 0])
```

```
array([-1, -1, 1, -1, 2])
```

```
np.random.choice(a, 2, False, p=[0.1, 0.1, 0, 0.8, 0]) # without replacement
```

```
array([-1, 2])
```

# Random Permutations

- ▶ There are two methods for permuting the contents of an array:
  - ▶ **np.random.shuffle()** randomly rearranges the order of the elements *in place*
  - ▶ **np.random.permutation()** makes a copy of the array first, leaving the original unchanged

```
a = np.arange(6)
np.random.permutation(a)

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

```
a

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

```
np.random.shuffle(a)
a

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

# Random Permutations

- ▶ These methods only act on the first dimension of the array:

```
a = np.arange(6).reshape(3, 2)
a
```

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

```
np.random.permutation(a) # permutes only the rows
```

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

# Sampling NonUniform Distributions

- The full range of random distributions supported by NumPy can be found [here](#)

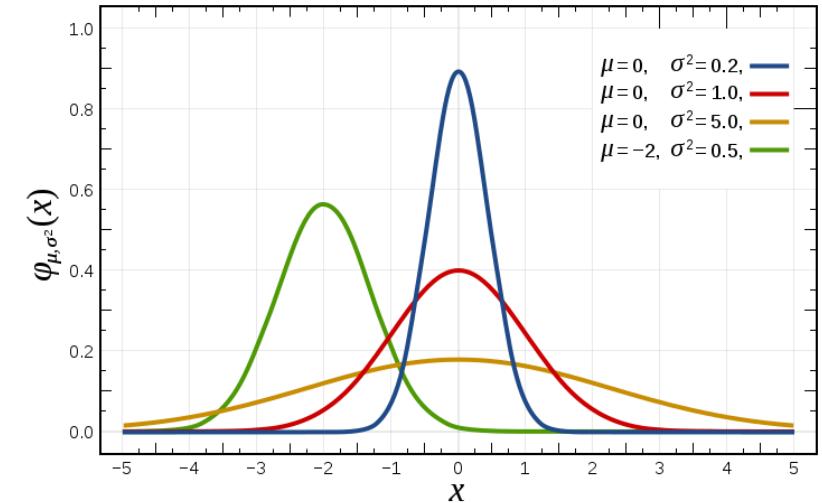
| Function                                                  | Description                                                                    |
|-----------------------------------------------------------|--------------------------------------------------------------------------------|
| <code>binomial(n, p[, size])</code>                       | Draw samples from a binomial distribution                                      |
| <code>chisquare(df[, size])</code>                        | Draw samples from a chi-square distribution                                    |
| <code>dirichlet(alpha[, size])</code>                     | Draw samples from the Dirichlet distribution                                   |
| <code>exponential([scale, size])</code>                   | Draw samples from an exponential distribution                                  |
| <code>gamma(shape[, scale, size])</code>                  | Draw samples from a Gamma distribution                                         |
| <code>geometric(p[, size])</code>                         | Draw samples from the geometric distribution                                   |
| <code>hypergeometric(ngood, nbad, nsample[, size])</code> | Draw samples from a Hypergeometric distribution                                |
| <code>logistic([loc, scale, size])</code>                 | Draw samples from a logistic distribution                                      |
| <code>multinomial(n, pvals[, size])</code>                | Draw samples from a multinomial distribution                                   |
| <code>normal([loc, scale, size])</code>                   | Draw random samples from a normal (Gaussian) distribution                      |
| <code>poisson([lam, size])</code>                         | Draw samples from a Poisson distribution                                       |
| <code>power(a[, size])</code>                             | Draws samples in [0, 1] from a power distribution with positive exponent a - 1 |

# The Normal Distribution

- ▶ The normal probability distribution is described by the Gaussian function:

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

- ▶ where  $\mu$  is the mean and  $\sigma$  the standard deviation



- ▶ **`np.random.normal(loc, scale, size)`** selects random samples from the normal distribution
  - ▶ The mean and standard deviation are specified by *loc* and *scale* respectively, which default to 0 and 1
  - ▶ The shape of the returned array is specified with the *size* attribute

# The Normal Distribution

## ► Examples:

```
np.random.normal()
```

```
-0.13021860515282227
```

```
np.random.normal(scale=5, size=3)
```

```
array([1.3220852 , 1.64185191, 2.8419821])
```

```
np.random.normal(100, 8, size=(4,2))
```

```
array([[108.96718853, 116.73807873],
 [102.50323123, 103.73690431],
 [92.42192965, 103.18331888],
 [93.94766241, 108.20371354]])
```

# The Normal Distribution

- ▶ It is also possible to draw numbers from the standard normal distribution (that with  $\mu = 0$  and  $\sigma = 1$ ) with **np.random.randn()**
- ▶ Like random.rand, this takes the dimensions of an array as its arguments:

```
np.random.randn(2, 2)
```

```
array([[1.37711873, -1.1703141],
 [-0.86701367, 0.39239136]])
```

- ▶ Although np.random.randn() doesn't provide a way to set the mean and standard deviation explicitly, the standard distribution can be rescaled easily enough:

```
mu, sigma = 100, 8
mu + sigma * np.random.randn(4, 2)
```

```
array([[106.15287112, 93.94864554],
 [95.04633335, 106.96496925],
 [100.79360207, 83.65518883],
 [99.63704383, 100.80514466]])
```

# The Binomial Distribution

- ▶ The binomial probability distribution describes the number of particular outcomes in a sequence of  $n$  Bernoulli trials
- ▶ That is,  $n$  independent experiments, each of which can yield exactly two possible outcomes (e.g., yes/no, success/failure, heads/tails)
- ▶ If the probability of a single particular outcome (say, success) is  $p$ , the probability that such a sequence of trials yields exactly  $k$  such outcomes is

$$\binom{n}{k} p^k (1-p)^{n-k} = \frac{n!}{k!(n-k)!} p^k (1-p)^{n-k}$$

- ▶ For example, when a fair coin is tossed, the probability of it coming up heads each time is  $1/2$ . The probability of getting exactly three heads out of four tosses, is therefore  $4(0.5)(0.5)^3 = 0.25$ .

# The Binomial Distribution

- ▶ To sample from the binomial distribution described by parameters n and p, use **np.random.binomial(n, p)**
- ▶ Again, the shape of an array of samples can be specified with the third argument, *size*:

```
np.random.binomial(4, 0.5)
```

```
1
```

```
np.random.binomial(4, 0.5, (4, 4))
```

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

# The Poisson Distribution

- ▶ The Poisson distribution describes the probability of a particular number of independent events occurring in a given interval of time if these events occur at a known average rate
- ▶ It is also used for occurrences in specified intervals over other domains such as distance or volume
- ▶ The Poisson probability distribution of the number of events,  $k$ , is

$$P(k) = \frac{\lambda^k e^{-\lambda}}{k!}$$

- ▶ where the parameter  $\lambda$  is the expected (average) number of events occurring within the considered interval

# The Poisson Distribution

- ▶ The NumPy implementation `np.random.poisson()` takes  $\lambda$  as its first argument (which defaults to 1) and, as before the shape of the desired array of samples can be specified with a second argument, size
- ▶ For example, if I receive an average of 2.5 emails an hour, a sample of the number of emails I receive each hour over the next 8 hours could be obtained as:

```
np.random.poisson(2.5, 8)
array([3, 2, 0, 1, 3, 0, 2, 1])
```

## Exercise

- ▶ Suppose an  $n$ -page book is known to contain  $m$  misprints
- ▶ If the misprints are independent of one another, the probability of a misprint occurring on a particular page is  $p = 1/n$  and their distribution may be considered to be binomial
- ▶ Write a short program to conduct a number of trial virtual “printings” of a book with  $n = 500$ ,  $m = 400$  and determine the probability,  $\Pr$ , that a single given page will contain two or more misprints
- ▶ Compare with the result predicted by the Poisson distribution with rate parameter  $\lambda = m/n$ ,

$$\Pr = 1 - e^{-\lambda} \left( \frac{\lambda^0}{0!} + \frac{\lambda^1}{1!} \right)$$

# Loading a Text File

- ▶ **np.loadtxt()** can be used to read a text file into a NumPy array

```
np.loadtxt(fname, dtype=<class 'float'>, comments='#', delimiter=None, converters=None, skiprows=0,
usecols=None, unpack=False, ndmin=0)
```

| Argument   | Description                                                                                                                                                   |
|------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| fname      | A filename, the only required argument                                                                                                                        |
| dtype      | The data type of the resulting array (defaults to float)                                                                                                      |
| comments   | Tells NumPy to ignore the contents of any line following this character (default is #)                                                                        |
| delimiter  | The string used to separate columns of data in the file. By default it is None meaning that any amount of whitespace (spaces, tabs) delimits the data.        |
| converters | An optional dictionary mapping the column index to a function converting string values in that column to data                                                 |
| skiprows   | An integer giving the number of lines at the start of the file to skip over before reading the data (default is 0)                                            |
| usecols    | A sequence of column indexes determining which columns of the file to return as data. By default it is None, meaning all columns will be parsed and returned. |

# Loading a Text File

- Consider the following text file of data relating to a (fictional) population of students
- This file can be downloaded as student-data.txt from <https://goo.gl/KnFPfC>

```
Student data collected on 17 July 2014
Researcher: Dr Wicks, University College Newbury
```

```
The following data relate to N = 20 students. It
has been totally made up and so therefore is 100%
anonymous.
```

| Subject<br>(ID) | Sex<br>M/F | DOB<br>dd/mm/yy | Height<br>m | Weight<br>kg | BP<br>mmHg | VO2max<br>mL.kg-1.min-1 |
|-----------------|------------|-----------------|-------------|--------------|------------|-------------------------|
| JW-1            | M          | 19/12/95        | 1.82        | 92.4         | 119/76     | 39.3                    |
| JW-2            | M          | 11/1/96         | 1.77        | 80.9         | 114/73     | 35.5                    |
| JW-3            | F          | 2/10/95         | 1.68        | 69.7         | 124/79     | 29.1                    |
| JW-6            | M          | 6/7/95          | 1.72        | 75.5         | 110/60     | 45.5                    |
| # JW-7          | F          | 28/3/96         | 1.66        | 72.4         | 101/68     | -                       |
| JW-9            | F          | 11/12/95        | 1.78        | 82.1         | 115/75     | 32.3                    |
| JW-10           | F          | 7/4/96          | 1.60        | -            | -/-        | 30.1                    |
| JW-11           | M          | 22/8/95         | 1.72        | 77.2         | 97/63      | 48.8                    |
| JW-12           | M          | 23/5/96         | 1.83        | 88.9         | 105/70     | 37.7                    |
| JW-14           | F          | 12/1/96         | 1.56        | 56.3         | 108/72     | 26.0                    |
| JW-15           | F          | 1/6/96          | 1.64        | 65.0         | 99/67      | 35.7                    |
| JW-16           | M          | 10/9/95         | 1.63        | 73.0         | 131/84     | 29.9                    |
| JW-17           | M          | 17/2/96         | 1.67        | 89.8         | 101/76     | 40.2                    |
| JW-18           | M          | 31/7/96         | 1.66        | 75.1         | -/-        | -                       |
| JW-19           | F          | 30/10/95        | 1.59        | 67.3         | 103/69     | 33.5                    |
| JW-22           | F          | 9/3/96          | 1.70        | -            | 119/80     | 30.9                    |
| JW-23           | M          | 15/5/95         | 1.97        | 89.2         | 124/82     | -                       |
| JW-24           | F          | 1/12/95         | 1.66        | 63.8         | 100/78     | -                       |
| JW-25           | F          | 25/10/95        | 1.63        | 64.4         | -/-        | 28.0                    |
| JW-26           | M          | 17/4/96         | 1.69        | -            | 121/82     | 39.                     |

# Loading a Text File

- ▶ Let's find the average heights of the male and female students
- ▶ The columns we need are the second and fourth
- ▶ First construct a record dtype for the two fields, then read the relevant columns after skipping the first nine header lines:

```
fname = "student-data.txt"
dtype1 = np.dtype([("gender", "S1"), ("height", "f8")])
students = np.loadtxt(fname, dtype=dtype1, skiprows=9, usecols=(1,3))
students

array([(b'M', 1.82), (b'M', 1.77), (b'F', 1.68), (b'M', 1.72),
 (b'F', 1.78), (b'F', 1.6), (b'M', 1.72), (b'M', 1.83),
 (b'F', 1.56), (b'F', 1.64), (b'M', 1.63), (b'M', 1.67),
 (b'M', 1.66), (b'F', 1.59), (b'F', 1.7), (b'M', 1.97),
 (b'F', 1.66), (b'F', 1.63), (b'M', 1.69)],
 dtype=[('gender', 'S1'), ('height', '<f8')])
```

# Loading a Text File

- To find the average heights of the male students, we only want to index the records with the gender field as M, for which we can create a boolean array:

```
males = students["gender"] == b'M'
males

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

- Bytes literals are always prefixed with 'b' or 'B'; they produce an instance of the bytes type instead of the str type. They may only contain ASCII characters.
- m has entries that are True or False for each of the 19 valid records (one is commented out) according to whether the student is male or female
- So the heights of the male students can be seen to be:

```
print(students["height"][males])

[1.82 1.77 1.72 1.72 1.83 1.63 1.67 1.66 1.97 1.69]
```

# Loading a Text File

- ▶ Therefore, the averages we need are:

```
male_avg = students["height"][males].mean()
female_avg = students["height"][~males].mean()
print("Male average: {:.2f}m, Female average: {:.2f}m"
 .format(male_avg, female_avg))
```

Male average: 1.75m, Female average: 1.65m

- ▶ Note that `~males` ("not males") is the inverse boolean array of `males`
- ▶ To perform the same analysis on the student weights we have a bit more work to do because there are some missing values (denoted by `'-'`)
- ▶ We'll write a converter method that will replace the missing values with `-99`
- ▶ The function `parse_weight()` expects a string argument and returns a float:

```
def parse_weight(s):
 try:
 return float(s)
 except ValueError:
 return -99
```

# Loading a Text File

- ▶ This is the function we want to pass as a converter for column 4:

```
dtype2 = np.dtype([("gender", "S1"), ("weight", "f8")])
students = np.loadtxt(fname, dtype=dtype2, skiprows=9, usecols=(1,4),
 converters={4: parse_weight})
```

- ▶ Now mask off the invalid data and index the array with a boolean array as before:

```
valid = students["weight"] > 0
male_avg = students["weight"][valid & males].mean()
female_avg = students["weight"][valid & ~males].mean()
print("Male average: {:.2f} kg, Female average: {:.2f} kg"
 .format(male_avg, female_avg))
```

Male average: 82.44 kg, Female average: 66.94 kg

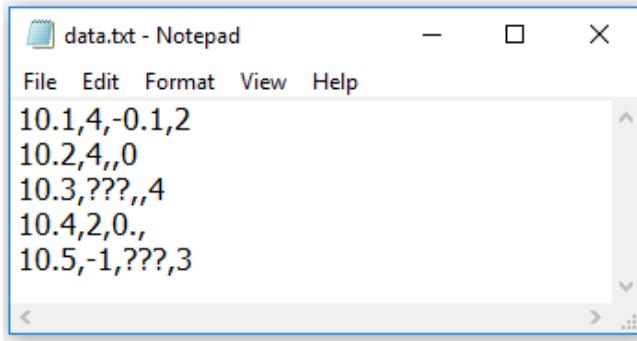
# Loading a Text File

- ▶ **np.genfromtxt()** function is similar to np.loadtxt(), but has a few more options and is able to cope with missing data
- ▶ If a data set is incomplete, np.loadtxt() will be unable to parse the fields with missing data into valid values for the array and will raise an exception
- ▶ np.genfromtxt(), however, sets missing or invalid entries equal to the default values given in the following table:

| Data type | Default value |
|-----------|---------------|
| int       | -1            |
| float     | np.nan        |
| bool      | False         |
| complex   | np.nan + 0.j  |

# Loading a Text File

- For example, the comma-separated file here has two ways of indicating missing data: empty fields and entries with '????':



- Accordingly, np.genfromtxt() sets the missing fields to its defaults:

```
data = np.genfromtxt("data.txt", dtype="f8, i4, f8, i4", delimiter=',')
print(data)
```

```
[(10.1, 4, -0.1, 2) (10.2, 4, nan, 0) (10.3, -1, nan, 4)
 (10.4, 2, 0., -1) (10.5, -1, nan, 3)]
```

## Loading a Text File

- ▶ The **missing\_values** and **filling\_values** arguments allow closer control over which default values to use for which columns
- ▶ If `missing_values` is given as a sequence of strings, each string is associated with a column in the data file, in order
- ▶ If given as a dictionary of string values, the keys denote either column indexes (if they are integers) or column names (if they are strings)
- ▶ The corresponding argument, `filling_values`, maps these column indexes or names to default values
- ▶ If `filling_values` is provided as a single value, this value is used for missing data in all columns

# Loading a Text File

- For example, to replace the invalid values in column 1 (indicated by '??') with 999, the missing or invalid values in column 2 (also indicated by '??') with -99 and the missing values in column 3 with 0:

```
data = np.genfromtxt("data.txt", dtype="f8, i4, f8, i4", delimiter=',',
 missing_values={1: '??', 2: '??'},
 filling_values={1: 999, 2: -99, 3: 0})
print(data)

[(10.1, 4, -0.1, 2) (10.2, 4, -99. , 0) (10.3, 999, -99. , 4)
 (10.4, 2, 0. , 0) (10.5, -1, -99. , 3)]
```

## Exercise

- ▶ The text file mountain-data.txt (available in <https://goo.gl/KnFPfC>) contains a list of the 14 highest mountains in the world with their names, height, year of first ascent, year of first winter ascent, and location as longitude and latitude in degrees
- ▶ Use NumPy to read these data into a suitable structured array to determine the following:
  - ▶ The lowest 8,000 m peak
  - ▶ The most northely, easterly, southerly and westerly peaks
  - ▶ The most recent first ascent of the peaks
  - ▶ The first of the peaks to be climbed in winter
- ▶ Also produce another structured array containing a list of mountains with their height in feet and first ascent date, ordered by increasing height, and save this array to a new text file
  - ▶ 1 metre = 3.2808399 feet

# Statistical Methods

- ▶ NumPy provides several methods for performing statistical analysis
- ▶ The full list of methods can be found [here](#)
- ▶ The following table shows the available order statistics methods:

| Function                            | Description                                                                              |
|-------------------------------------|------------------------------------------------------------------------------------------|
| amin(a[, axis, out, keepdims])      | Return the minimum of an array or minimum along an axis<br>np.min is an alias to np.amin |
| amax(a[, axis, out , keepdims])     | Return the maximum of an array or maximum along an axis<br>np.max is an alias to np.amax |
| nanmin(a[, axis, o ut, keepdims])   | Return minimum of an array or minimum along an axis, ignoring any NaNs                   |
| nanmax(a[, axis, out, keepdims])    | Return the maximum of an array or maximum along an axis, ignoring any NaNs               |
| ptp(a[, axis, out])                 | Range of values (maximum - minimum) along an axis                                        |
| percentile(a, q[, a xis, out, ...]) | Compute the qth percentile of the data along the specified axis                          |

# Order Statistics

- ▶ The function **np.percentile()** returns a specified percentile, q, of the data along an axis (or along a flattened version of the array if no axis is given)
  - ▶ The minimum of an array is the value at q=0 (0th percentile), the maximum is at q=100 (100th percentile) and the median is at q=50 (50th percentile)
  - ▶ If no single value in the array corresponds to q exactly, a weighted average of the two nearest values is used

```
a = np.array([[0., 0.6, 1.2], [1.8, 2.4, 3.0]])
np.percentile(a, 50)
```

1.5

```
np.percentile(a, 50, axis=1)
```

array([0.6, 2.4])

```
np.percentile(a, 75, axis=1)
```

array([0.9, 2.7])

# Average and Variances

- ▶ NumPy provides the following functions for computing averages and variances:

| Function                                          | Description                                                                  |
|---------------------------------------------------|------------------------------------------------------------------------------|
| median(a[, axis, out, overwrite_input, keepdims]) | Compute the median along the specified axis                                  |
| average(a[, axis, weights, returned])             | Compute the weighted average along the specified axis                        |
| mean(a[, axis, dtype, out, keepdims])             | Compute the arithmetic mean along the specified axis                         |
| std(a[, axis, dtype, out, ddof, keepdims])        | Compute the standard deviation along the specified axis                      |
| var(a[, axis, dtype, out, ddof, keepdims])        | Compute the variance along the specified axis                                |
| nanmean(a[, axis, dtype, out, keepdims])          | Compute the arithmetic mean along the specified axis, ignoring NaNs          |
| nanstd(a[, axis, dtype, out, ddof, keepdims])     | Compute the standard deviation along the specified axis, while ignoring NaNs |

# Averages

- ▶ **np.mean()** calculates the arithmetic mean of the values of an array, while **np.average()** is used to calculate a weighted average
- ▶ The weighted average is calculated as: 
$$\bar{x}_w = \frac{\sum_{i=1}^n w_i x_i}{\sum_{i=1}^n w_i}$$
- ▶ where the weights,  $w_i$ , are supplied as a sequence of the same length as the array

```
a = np.array([1, 4, 9, 16])
np.mean(a)
```

7.5

```
np.average(a, weights=[0, 3, 1, 0]) # (3 * 4 + 1 * 9) / (3 + 1)
```

5.25

```
Computing a weighted average along the second axis
a = np.array([[1., 8., 27], [-0.5, 1., 0.]])
np.average(a, weights=[0, 1, 0.1], axis=1)
```

```
array([9.72727273, 0.90909091])
```

# Standard Deviation and Variance

- ▶ `np.std()` calculates, by default, the **uncorrected sample standard deviation**:

$$\sigma_n = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2}$$

- ▶ where  $x_i$  are the  $N$  observed values in the array and  $\bar{x}$  is their mean
- ▶ To calculate the **corrected sample standard deviation**:

$$\sigma_n = \sqrt{\frac{1}{N-\delta} \sum_{i=1}^N (x_i - \bar{x})^2}$$

- ▶ pass to the argument `ddof` the value of  $\delta$  such that  $N - \delta$  is the number of degrees of freedom in the sample
- ▶ In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of the infinite population

# Standard Deviation and Variance

- ▶ For example:

```
x = np.array([1, 2, 3, 4])
np.std(x) # or x.std(), uncorrected standard deviation
```

1.118033988749895

```
np.std(x, ddof=1) # corrected standard deviation
```

1.2909944487358056

- ▶ The function **np.nanstd()** calculates the standard deviation ignoring np.nan values (so that  $N$  is the number of non-NaN values in the array)
- ▶ NumPy also has methods for calculating the variance of the values in an array: **np.var()** and **np.nanvar()**

# Covariance

- ▶ **np.cov(X)** returns the covariance of X
- ▶ X is a two-dimensional array, in which the rows represent variables,  $x_i$ , and the columns represent observations of the value of each variable
- ▶ np.cov(X) returns the covariance matrix,  $C_{ij}$ , indicating how variable  $x_i$  varies with  $x_j$ :

$$C_{ij} \equiv \text{cov}(x_i, x_j) = E[(x_i - \mu_i)(x_j - \mu_j)]$$

- ▶ where  $\mu_i$  is the mean of the variable  $x_i$
- ▶ By default, np.cov(X) returns the **unbiased** estimate of the covariance:

$$C_{ij} = \frac{1}{N-1} \sum_{k=1}^N (x_{ik} - \mu_i)(x_{jk} - \mu_j) \quad \mu_i = \frac{1}{N} \sum_{k=1}^N x_{ik}$$

- ▶ but if the bias argument is set to 1, then N is used in the denominator instead of  $N - 1$  to give the **biased** estimate of the covariance

# Covariance

- As an example, consider the covariance of the matrix X, consisting of five observations of three variables,  $x_0$ ,  $x_1$  and  $x_2$ :

```
X = np.array([[0.1, 0.3, 0.4, 0.8, 0.9],
 [3.2, 2.4, 2.4, 0.1, 5.5],
 [10., 8.2, 4.3, 2.6, 0.9]
])
```

```
print(np.cov(X))
```

```
[[0.115 0.0575 -1.2325]
 [0.0575 3.757 -0.8775]
 [-1.2325 -0.8775 14.525]]
```

- This matrix indicates a strong anticorrelation between  $x_0$  and  $x_2$  (as one increases the other decreases) and no strong correlation between  $x_0$  and  $x_1$  ( $x_0$  and  $x_1$  don't trend strongly together)
- The diagonal elements  $C_{ii}$  are the unbiased estimators of the variances of  $x_i$

```
print(np.var(X, axis=1, ddof=1))
```

```
[0.115 3.757 14.525]
```

# Correlation Coefficient Matrix

- The *correlation coefficient matrix* is often used instead of the covariance matrix as it is normalized by dividing  $C_{ij}$  by the product of the variables' standard deviations:

$$P_{ij} = \text{corr}(x_i, x_j) = \frac{C_{ij}}{\sigma_i \sigma_j} = \frac{C_{ij}}{\sqrt{C_{ii} C_{jj}}}$$

- This means the elements  $P_{ij}$  have values between -1 and 1, inclusive, and the diagonal elements are  $P_{ii} = 1$
- $P_{ij}$  can be computed by calling **np.corrcoef(X)**:

```
print(np.corrcoef(X))
```

```
[[1. 0.0874779 -0.95363007]
 [0.0874779 1. -0.11878687]
 [-0.95363007 -0.11878687 1.]]
```

- It is easy to see the strong anticorrelation between  $x_0$  and  $x_2$  ( $C_{0,2} = -0.954$ ) and the lack of correlation between  $x_1$  and the other variables (e.g.,  $C_{1,0} = 0.087$ )

# Correlation Coefficient Matrix

- Both np.cov() and np.corrcoef() can take a second array containing a further set of variables and observations:

```
x = np.array([1, 2, 3, 4, 5])
y = np.array([0.08, 0.31, 0.41, 0.48, 0.62])
print(np.corrcoef(x, y))
```

```
[[1. 0.97787645]
 [0.97787645 1.]]
```

- This is a convenient alternative to:

```
print(np.corrcoef(np.vstack((x, y))))
```

```
[[1. 0.97787645]
 [0.97787645 1.]]
```

- If your observations happen to be in the rows of your matrix, with the variables corresponding to the columns, you can pass rowvar=0 to either np.cov or np.corrcoef

## Exercise

- ▶ The Cambridge University Digital Technology Group have been recording the weather from the roof of their department building since 1995 and make the data available to download in a single CSV file at [www.cl.cam.ac.uk/research/dtg/weather](http://www.cl.cam.ac.uk/research/dtg/weather)
- ▶ Write a program to determine the correlation coefficient between temperature and pressure at this site
- ▶ Note that some lines have missing temperatures or pressure measurements
  - ▶ e.g, line no. 55485: 1998-09-09 01:00:00,,,156,998,47,180,0,0,140

# Linear Algebra

- ▶ NumPy contains many linear algebra functions, including:
  - ▶ Matrix operations
  - ▶ Eigenvalues and eigenvectors
  - ▶ Solving systems of equations
- ▶ Some of these functions reside in the **numpy.linalg** module

# Basic Matrix Operations

- ▶ All the matrix operations can be carried out on both a regular two-dimensional NumPy array and on NumPy matrix object (to be discussed later)
- ▶ Example for matrix addition:

```
A = np.array([[0, 0.5], [-1, 2]])
A
```

```
array([[0. , 0.5],
 [-1. , 2.]])
```

```
B = np.array([[2, -0.5], [3, 1.5]])
B
```

```
array([[2. , -0.5],
 [3. , 1.5]])
```

```
A + B # elementwise addition
```

```
array([[1. , 1.5],
 [1. , 4.]])
```

# Matrix and Vector Products

| Function                | Description                                                                  |
|-------------------------|------------------------------------------------------------------------------|
| dot(a, b[, out])        | Dot product of two arrays                                                    |
| vdot(a, b)              | Return the dot product of two vectors                                        |
| matmul(a, b[, out])     | Matrix product of two arrays                                                 |
| inner(a, b)             | Inner product of two arrays                                                  |
| tensordot(a, b[, axes]) | Compute tensor dot product along specified axes for arrays $\geq 1\text{-D}$ |
| outer(a, b[, out])      | Compute the outer product of two vectors                                     |

# Matrix and Vector Products

- ▶ **np.dot(a, b)** computes the dot product of arrays a and b:
  - ▶ If both *a* and *b* are 1-D arrays, it is the inner product of vectors
  - ▶ If either *a* or *b* is 0-D (scalar), it is equivalent to multiplication and using np.multiply(a, b) or *a* \* *b* is preferred
  - ▶ Otherwise it is matrix multiplication, but using np.matmul(a, b) or *a* @ *b* is preferred

```
A = np.array([[0, 0.5], [-1, 2]])
B = np.array([[2, -0.5], [3, 1.5]])
A.dot(B)
```

```
array([[1.5 , 0.75],
 [4. , 3.5]])
```

```
A @ B # the same as A.dot(B)
```

```
array([[1.5 , 0.75],
 [4. , 3.5]])
```

```
x = np.array([1, 1])
A.dot(x) # Ax: matrix and vector product
```

```
array([0.5, 1.])
```

# Matrix and Vector Products

- ▶ **np.vdot(a, b)** computes the dot product of arrays a and b:
- ▶ **vdot(a, b)** handles complex numbers differently than **dot(a, b)**
  - ▶ If the first argument is complex the complex conjugate of the first argument is used for the calculation of the dot product
- ▶ **vdot(a, b)** handles multidimensional arrays differently than **dot(a, b)**: it doesn't perform a matrix product, but flattens input arguments to 1-D vectors first

```
a = np.array([[1, 4], [5, 6]])
b = np.array([[4, 1], [2, 2]])
np.vdot(a, b) # 1*4 + 4*1 + 5*2 + 6*2
```

30

# Matrix and Vector Products

- ▶ `np.matmul(a, b, out=None)` computes a matrix product of two arrays
- ▶ The behavior depends on the arguments in the following way:
  - ▶ If both arguments are 2-D they are multiplied like conventional matrices
  - ▶ If either argument is N-D,  $N > 2$ , it is treated as a stack of matrices residing in the last two indexes and broadcast accordingly
  - ▶ If the first argument is 1-D, it is promoted to a matrix by prepending a 1 to its dimensions.  
After matrix multiplication the prepended 1 is removed
  - ▶ If the second argument is 1-D, it is promoted to a matrix by appending a 1 to its dimensions.  
After matrix multiplication the appended 1 is removed
- ▶ `matmul` differs from `dot` in two important ways:
  - ▶ Multiplication by scalars is not allowed
  - ▶ Stacks of matrices are broadcast together as if the matrices were elements
- ▶ `matmul` implements the semantics of the `@` operator introduced in Python 3.5

# Matrix and Vector Products

```
a = [[1, 0], [0, 1]]
b = [[4, 1], [2, 2]]
np.matmul(a, b)
```

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

```
a = [[1, 0], [0, 1]]
b = [1, 2]
np.matmul(a, b)
```

```
array([1, 2])
```

```
np.matmul(b, a)
```

```
array([1, 2])
```

```
np.matmul([1, 2], 3)
```

```


ValueError Traceback (most recent
call last)
<ipython-input-110-4a3e172e0eeb> in <module>()
----> 1 np.matmul([1, 2], 3)
```

```
ValueError: Scalar operands are not allowed, use '*' instead
```

# Matrix and Vector Products

- ▶ **np.inner(a, b)** computes the inner product of two arrays
  - ▶ For vectors (1-D arrays) it computes the ordinary inner-product
  - ▶ In higher dimensions it computes a sum product over the last axes:

```
np.inner(a, b)[i0,...,ir-1,j0,...,js-1]
= sum(a[i0,...,ir-1,:]*b[j0,...,js-1,:])
```

- ▶ The shape of the result is *out.shape = a.shape[:-1] + b.shape[:-1]*

```
a = np.array([[0, 0.5], [-1, 2]])
b = np.array([[2, -0.5], [3, 1.5]])
np.inner(a, b) # [[0*2+0.5*(-0.5), 0*3+0.5*1.5],
[-1*2+2*(-0.5), (-1)*3+2*1.5]]
```

```
array([[-0.25, 0.75],
 [-3. , 0.]])
```

```
a = np.arange(24).reshape((2, 3, 4))
b = np.arange(4)
np.inner(a, b)
```

```
array([[14, 38, 62],
 [86, 110, 134]])
```

# Matrix Exponentiation

- ▶ To raise a matrix to an (integer) power requires a function from the np.linalg module:

```
A = np.array([[0, 0.5], [-1, 2]])
np.linalg.matrix_power(A, 3) # the same as A.dot(A.dot(A))
```

```
array([[-1. , 1.75],
 [-3.5, 6.]])
```

- ▶ On the other hand, the `**` operator performs *elementwise* exponentiation:

```
A**3 # the same as A * A * A
```

```
array([[0. , 0.125],
 [-1. , 8.]])
```

# Matrix Transpose

- ▶ To find the transpose of a matrix, use the **transpose()** method

```
A = np.array([[0, 0.5], [-1, 2]])
A.transpose() # or simply A.T
```

```
array([[0., -1.],
 [0.5, 2.]])
```

- ▶ Note that the transpose returns a *view* on the original matrix

# Trace

- ▶ The **trace** of an n-by-n square matrix  $A$  is the sum of the elements on the main diagonal, i.e.,

$$tr(A) = \sum_{i=1}^n a_{ii} = a_{11} + a_{22} + \dots + a_{nn}$$

- ▶ The trace of a matrix is the sum of its (complex) eigenvalues
- ▶ The **trace()** method of a matrix returns its trace:

```
A = np.array([[1, 2], [3, 4]])
A.trace()
```

5

# Matrix Rank

- ▶ The **rank** of a matrix  $A$  is the dimension of the vector space generated by its columns
- ▶ This corresponds to the maximal number of linearly independent columns of  $A$
- ▶ The column rank and the row rank are always equal
- ▶ A common approach to finding the rank of a matrix is to reduce it to a simpler form, generally row echelon form, by elementary row operations
  - ▶ Row operations don't change the row space (hence don't change the row rank)
- ▶ Once in row echelon form, the rank equals the number of non-zero rows.
- ▶ The rank of a matrix is obtained using **np.linalg.matrix\_rank()**:

```
A = np.array([[0, 0.5], [-1, 2]])
np.linalg.matrix_rank(A) # matrix A has full rank
```

2

```
D = np.array([[1, 1], [2, 2]]) # a rank deficient matrix
np.linalg.matrix_rank(D)
```

1

# Determinant

- ▶ The **determinant** of a matrix  $A$  is denoted  $\det(A)$ ,  $\det A$ , or  $|A|$
- ▶ Geometrically, it can be viewed as the scaling factor of the linear transformation  $A$
- ▶ In the case of a  $2 \times 2$  matrix the determinant is defined as:

$$|A| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

- ▶ Similarly, for a  $3 \times 3$  matrix  $A$ , its determinant is:

$$\begin{aligned} |A| &= \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} \square & \square & \square \\ \square & e & f \\ \square & h & i \end{vmatrix} - b \begin{vmatrix} \square & \square & \square \\ d & \square & f \\ g & \square & i \end{vmatrix} + c \begin{vmatrix} \square & \square & \square \\ d & e & \square \\ g & h & \square \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix} \\ &= aei + bfg + cdh - ceg - bdi - afh. \end{aligned}$$

- ▶ Each determinant of a  $2 \times 2$  matrix in this equation is called a **minor** of the matrix  $A$
- ▶ This procedure can be extended to give a recursive definition for the determinant of an  $n \times n$  matrix, the *minor expansion formula*

# Determinant

- ▶ For example, let's compute the determinant of

$$A = \begin{pmatrix} 1 & 6 & 4 \\ 2 & 7 & 3 \\ 8 & 9 & 5 \end{pmatrix}$$

$$\det(A) = 1 \cdot (7 \cdot 5 - 3 \cdot 9) - 6 \cdot (2 \cdot 5 - 3 \cdot 8) + 4 \cdot (2 \cdot 9 - 7 \cdot 8) = 8 + 84 - 152 = -60$$

- ▶ The function **np.linalg.det()** returns the determinant of a matrix:

```
A = np.array([[1, 6, 4], [2, 7, 3], [8, 9, 5]])
np.linalg.det(A)
```

-59.99999999999986

# Matrix Norm

- ▶ The norm of a matrix or vector is returned by the function **np.linalg.norm()**
- ▶ It is possible to calculate several different norms (see the [documentation](#))
- ▶ The ones used by default are:

▶ Euclidean norm for one-dimensional arrays:  $\| a \| = \left( \sum_i a_i^2 \right)^{1/2} = \sqrt{a_0^2 + a_1^2 + \dots + a_{n-1}^2}$

▶ Frobenius norm for two-dimensional arrays:  $\| A \| = \left( \sum_{i,j} a_{ij}^2 \right)^{1/2}$

```
c = np.array([1, 6, 2])
np.linalg.norm(c) # sqrt(1 + 36 + 4)
```

6.4031242374328485

```
A = np.array([[0, 0.5], [-1, 2]])
np.linalg.norm(A)
```

2.29128784747792

# Matrix Inversion

- ▶ An  $n \times n$  matrix  $A$  is called **invertible** (or **nonsingular**) if there exists an  $n \times n$  matrix  $B$  such that  $AB = BA = I_n$ , where  $I_n$  denotes the  $n \times n$  identity matrix
- ▶ In this case, the matrix  $B$  is called the **inverse** of  $A$ , denoted by  $A^{-1}$
- ▶ A square matrix that is **not** invertible is called **singular** or **degenerate**
- ▶ A square matrix is singular if and only if its determinant is 0
- ▶ For example,  $A = \begin{pmatrix} -1 & \frac{3}{2} \\ 1 & -1 \end{pmatrix}$  is invertible, since its determinant is  $\det A = -1/2 \neq 0$
- ▶ However,  $B = \begin{pmatrix} -1 & \frac{3}{2} \\ \frac{2}{3} & -1 \end{pmatrix}$  is non-invertible, since its determinant is 0

# Matrix Inversion

- ▶ **Matrix inversion** is the process of finding the matrix  $B$  that satisfies the prior equation for a given invertible matrix  $A$
- ▶ A variant of Gaussian elimination called Gauss–Jordan elimination can be used for finding the inverse of a matrix
  - ▶ The  $n \times n$  identity matrix is augmented to the right of  $A$ , forming an  $n \times 2n$  block matrix  $[A \mid I]$
  - ▶ Through application of elementary row operations, find the reduced echelon form of this  $n \times 2n$  matrix
  - ▶ The matrix  $A$  is invertible if and only if the left block can be reduced to the identity matrix  $I$ 
    - ▶ In this case the right block of the final matrix is  $A^{-1}$

# Matrix Inversion

- For example, let's find the inverse of the matrix

$$A = \begin{pmatrix} 7 & 2 & 1 \\ 0 & 3 & -1 \\ -3 & 4 & -2 \end{pmatrix}$$

$$\left( \begin{array}{ccc|ccc} 7 & 2 & 1 & 1 & 0 & 0 \\ 0 & 3 & -1 & 0 & 1 & 0 \\ -3 & 4 & -2 & 0 & 0 & 1 \end{array} \right) \xrightarrow{R_3 + \frac{3}{7}R_1 \rightarrow R_3} \left( \begin{array}{ccc|ccc} 7 & 2 & 1 & 1 & 0 & 0 \\ 0 & 3 & -1 & 0 & 1 & 0 \\ 0 & \frac{34}{7} & -\frac{11}{7} & \frac{3}{7} & 0 & 1 \end{array} \right) \xrightarrow{R_3 - \frac{34}{21}R_2 \rightarrow R_3} \left( \begin{array}{ccc|ccc} 7 & 2 & 1 & 1 & 0 & 0 \\ 0 & 3 & -1 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{21} & \frac{3}{7} & -\frac{34}{21} & 1 \end{array} \right) \xrightarrow{R_1 - \frac{2}{3}R_2 \rightarrow R_1}$$
  

$$\left( \begin{array}{ccc|ccc} 7 & 0 & \frac{5}{3} & 1 & -\frac{2}{3} & 0 \\ 0 & 3 & -1 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{21} & \frac{3}{7} & -\frac{34}{21} & 1 \end{array} \right) \xrightarrow{21R_3 \rightarrow R_3} \left( \begin{array}{ccc|ccc} 7 & 0 & \frac{5}{3} & 1 & -\frac{2}{3} & 0 \\ 0 & 3 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 9 & -34 & 21 \end{array} \right) \xrightarrow{\begin{matrix} R_1 - \frac{5}{3}R_3 \rightarrow R_1 \\ R_2 + R_3 \rightarrow R_2 \end{matrix}} \left( \begin{array}{ccc|ccc} 7 & 0 & 0 & -14 & 56 & -35 \\ 0 & 3 & 0 & 9 & -33 & 21 \\ 0 & 0 & 1 & 9 & -34 & 21 \end{array} \right) \xrightarrow{\begin{matrix} \frac{1}{7}R_1 \rightarrow R_1 \\ \frac{1}{3}R_2 \rightarrow R_2 \end{matrix}} \left( \begin{array}{ccc|ccc} 1 & 0 & 0 & -2 & 8 & -5 \\ 0 & 1 & 0 & 3 & -11 & 7 \\ 0 & 0 & 1 & 9 & -34 & 21 \end{array} \right)$$

- Thus,  $A^{-1} = \begin{pmatrix} -2 & 8 & -5 \\ 3 & -11 & 7 \\ 9 & -34 & 21 \end{pmatrix}$

# Matrix Inversion

- ▶ To find the inverse of a square matrix in NumPy, use **np.linalg.inv()**:

```
A = np.array([[7, 2, 1], [0, 3, -1], [-3, 4, -2]])
np.linalg.inv(A)
```

```
array([[-2., 8., -5.],
 [3., -11., 7.],
 [9., -34., 21.]])
```

# Matrix Inversion

- ▶ To find the inverse of a square matrix, use `np.linalg.inv()`:

```
A = np.array([[0, 0.5], [-1, 2]])
np.linalg.inv(A)
```

```
array([[4., -1.],
 [2., 0.]])
```

- ▶ A LinAlgError exception is raised if the matrix inversion fails:

```
B = np.array([[1, 1], [2, 2]])
np.linalg.inv(B)
```

```
LinAlgError Traceback (most recent call last)
<ipython-input-14-ad8e45c0361e> in <module>()
 1 B = np.array([[1, 1], [2, 2]])
----> 2 np.linalg.inv(B)
```

## Exercise

- ▶ Create the following matrix:

$$G = \begin{pmatrix} 10 & 40 & 20 \\ 6 & 20 & 1 \\ 3 & 2 & 0 \end{pmatrix}$$

- ▶ Find the transpose of  $G$
- ▶ Compute the rank of  $G$  (first by hand, then in numpy)
- ▶ Compute the determinant of  $G$  (first by hand, then in numpy)
- ▶ Compute the inverse of  $G$  (first by hand, then in numpy)
- ▶ Compute the condition number of  $G$  (first by hand, then in numpy)

# Pandas

Lecturer: Ben Galili

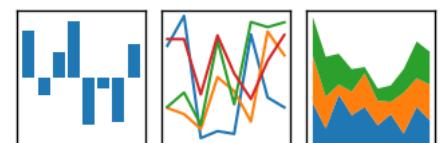


# Pandas

- ▶ *pandas* is a high-performance, open source library for data analysis in Python
  - ▶ Developed by Wes McKinney in 2008
- ▶ Key features of pandas:
  - ▶ It can process a variety of data sets in different formats: time series, tabular heterogeneous, and matrix data
  - ▶ It facilitates loading/importing data from varied sources such as CSV and DB/SQL
  - ▶ It can handle a myriad of operations on data sets: subsetting, slicing, filtering, merging, groupBy, re-ordering, and re-shaping
  - ▶ It can be used for parsing and munging (conversion) of data as well as modeling and statistical analysis
  - ▶ It integrates well with other Python libraries such as SciPy, and scikit-learn
  - ▶ It delivers fast performance

**pandas**

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



# Importing pandas

- ▶ Typically the pandas library is imported under the alias pd:

```
import pandas as pd
```

- ▶ Typically, pandas will be imported together with NumPy:

```
import numpy as np
import pandas as pd
```

# Pandas Objects

- ▶ NumPy's ndarray data structure provides essential features for the type of clean, well-organized data typically seen in numerical computing tasks
- ▶ Its limitations become clear when we need more flexibility (e.g., attaching labels to data, working with missing data, etc.) and when attempting operations that do not map well to element-wise broadcasting (e.g., groupings, pivots, etc.)
- ▶ Pandas objects build on the NumPy array structure and provide efficient access to these sorts of "data munging" tasks that occupy much of a data scientist's time
- ▶ There are three main data structures in pandas:
  - ▶ Series
  - ▶ DataFrame
  - ▶ Panel

# Series

- ▶ A Series is a one-dimensional array of indexed data (of any NumPy data type)
  - ▶ You can think of it as a fixed-length, ordered dict, mapping index values to data values
- ▶ The syntax for creating a Series data structure is:

```
pd.Series(data, index=idx)
```

- ▶ The simplest Series is formed from only an array of data:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
data
```

```
0 0.25
1 0.50
2 0.75
3 1.00
dtype: float64
```

- ▶ The Series wraps both a sequence of values and a sequence of indices
- ▶ If an index is not specified, a default index [0,... n-1], is created

# Series Attributes

- ▶ We can access the sequence of values and the index object of the Series via its **values** and **index** attributes, respectively:

```
data.values
```

```
array([0.25, 0.5 , 0.75, 1.])
```

```
data.index
```

```
RangeIndex(start=0, stop=4, step=1)
```

- ▶ The values are simply a familiar NumPy array
- ▶ The index is an array-like object of type pd.Index

# Indexing and Slicing

- Like with a NumPy array, data can be accessed by the associated index via the familiar Python square-bracket notation:

```
data[1]
```

```
0.5
```

```
data[1:3]
```

```
1 0.50
2 0.75
dtype: float64
```

```
data[data > 0.5]
```

```
2 0.75
3 1.00
dtype: float64
```

# Series as a Generalized NumPy Array

- ▶ The essential difference between a NumPy array and a Series object is the presence of the index
- ▶ While the Numpy Array has an *implicitly defined* integer index used to access the values, the Pandas Series has an *explicitly defined* index associated with the values
- ▶ This explicit index definition gives the Series object additional capabilities
- ▶ The index need not be an integer, but can consist of values of any desired type
- ▶ For example, we can use strings as an index:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
 index=['a', 'b', 'c', 'd'])

data
```

```
a 0.25
b 0.50
c 0.75
d 1.00
dtype: float64
```

# Series as a Generalized NumPy Array

- ▶ And the item access works as expected:

```
data['b']
```

```
0.5
```

```
data['c'] = 0.8
data
```

```
a 0.25
b 0.50
c 0.80
d 1.00
dtype: float64
```

- ▶ We can even use non-contiguous or non-sequential indices:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
 index=[2, 5, 3, 7])
data
```

```
2 0.25
5 0.50
3 0.75
7 1.00
dtype: float64
```

# Series as a Generalized NumPy Array

- ▶ And the item access works as expected:

```
data['b']
```

```
0.5
```

```
data['c'] = 0.8
data
```

```
a 0.25
b 0.50
c 0.80
d 1.00
dtype: float64
```

- ▶ We can even use non-contiguous or non-sequential indices:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
 index=[2, 5, 3, 7])
data
```

```
2 0.25
5 0.50
3 0.75
7 1.00
dtype: float64
```

# Series as a Generalized NumPy Array

- ▶ A Series's index can be altered in place by assignment:

```
data.index = [1, 3, 5, 10]
data
```

```
1 0.25
3 0.50
5 0.75
10 1.00
dtype: float64
```

# Series as a Generalized NumPy Array

- ▶ A Series's index can be altered in place by assignment:

```
data.index = [1, 3, 5, 10]
data
```

```
1 0.25
3 0.50
5 0.75
10 1.00
dtype: float64
```

# Series as a Specialized Dictionary

- ▶ You can also think of a Pandas Series as a specialization of a Python dictionary
- ▶ A dictionary is a structure that maps arbitrary keys to a set of arbitrary values, while a Series is a structure which maps *typed* keys to a set of *typed* values
  - ▶ The type information makes Pandas Series more efficient than Python dictionaries for certain operations (just as NumPy arrays are more efficient than Python lists for certain operations)
- ▶ A Series object can be constructed directly from a Python dictionary:

```
population_dict = {'California': 39776830,
 'Texas': 28704330,
 'Florida': 21312211,
 'New York': 19862512,
 'Pennsylvania': 12823989}
population = pd.Series(population_dict)
population
```

```
California 39776830
Florida 21312211
New York 19862512
Pennsylvania 12823989
Texas 28704330
dtype: int64
```

# Series as a Specialized Dictionary

- ▶ By default, a Series will be created where the index is drawn from the sorted keys
- ▶ From here, typical dictionary-style item access can be performed:

```
population['California']
```

```
39776830
```

- ▶ Unlike a dictionary, the Series also supports array-style operations such as slicing:

```
population['California': 'New York']
```

```
California 39776830
Florida 21312211
New York 19862512
dtype: int64
```

- ▶ Slicing with labels behaves differently than normal Python slicing in that the endpoint is **inclusive**

# Series as a Specialized Dictionary

- ▶ By default, a Series will be created where the index is drawn from the sorted keys
- ▶ From here, typical dictionary-style item access can be performed:

```
population['California']
```

```
39776830
```

- ▶ Unlike a dictionary, the Series also supports array-style operations such as slicing:

```
population['California': 'New York']
```

```
California 39776830
Florida 21312211
New York 19862512
dtype: int64
```

- ▶ Slicing with labels behaves differently than normal Python slicing in that the endpoint is **inclusive**

# Creating a Series from a Dictionary

- ▶ The index can be explicitly set if a different result is preferred:

```
population = pd.Series(population_dict,
 index=['California', 'New York', 'Florida'])
population
```

```
California 39776830
New York 19862512
Florida 21312211
dtype: int64
```

- ▶ In this case, the Series is populated only with the explicitly identified keys
- ▶ Just as in the case of dict, KeyError is raised if you try to retrieve a missing label:

```
try:
 population['Illinois']
except KeyError:
 print("Country's data is not available")
```

```
Country's data is not available
```

## Exercise

- ▶ Create a Pandas series from each of the items below: a list, a NumPy array and a dictionary:

```
my_list = list('abcdefghijklmnopqrstuvwxyz')
my_arr = np.arange(26)
my_dict = dict(zip(my_list, my_arr))
```

# Operations on Series

- ▶ Just like NumPy arrays, you can apply math functions on Series:

```
ser2 * 2
```

```
d 12
b 14
a -10
c 6
dtype: int64
```

```
np.exp(ser2)
```

```
d 403.428793
b 1096.633158
a 0.006738
c 20.085537
dtype: float64
```

```
np.mean(ser2)
```

```
2.75
```

```
np.std(ser2)
```

```
4.710360920354193
```

# Detecting Missing Data

- ▶ The **isnull()** and **notnull()** functions in pandas can be used to detect missing data:

```
pd.isnull(ser4)
```

```
California True
Ohio False
Oregon False
Texas False
dtype: bool
```

```
pd.notnull(ser4)
```

```
California False
Ohio True
Oregon True
Texas True
dtype: bool
```

- ▶ We'll discuss handling missing data later in the course

# Alignment

- An important feature of Series is that the data is automatically differently-indexed data on the basis of the label:

```
ser3
```

```
Ohio 35000
Oregon 16000
Texas 71000
Utah 5000
dtype: int64
```

```
ser4
```

```
California NaN
Ohio 35000.0
Oregon 16000.0
Texas 71000.0
dtype: float64
```

```
ser3 + ser4
```

```
California NaN
Ohio 70000.0
Oregon 32000.0
Texas 142000.0
Utah NaN
dtype: float64
```

# DataFrame

- ▶ DataFrame is the most commonly used data structure in pandas
- ▶ A DataFrame represents a tabular, spreadsheet-like data structure containing an ordered collection of columns, each of which can be a different type (int, bool, etc.)
- ▶ DataFrame is an analog of a two-dimensional array with both flexible row indices and flexible column names
- ▶ It can be thought of as a dictionary of aligned Series objects, sharing the same index
  - ▶ A DataFrame column is a Series structure
- ▶ It is similar to structured arrays in NumPy with mutability added
  - ▶ Columns can be inserted and deleted

# DataFrame Creation

- ▶ There are numerous ways to construct a DataFrame
- ▶ The constructor accepts many different types of arguments:
  - ▶ Dictionary of 1D NumPy arrays, lists, dictionaries, or Series structures
  - ▶ 2D NumPy array
  - ▶ Structured NumPy array
  - ▶ Another DataFrame structure
  - ▶ and more
- ▶ Row label indexes and column labels can be specified along with the data
- ▶ If they're not specified, they will be generated from the input data in an intuitive way
  - ▶ For example, from the keys of dict (for column labels) or by using range(n) for row labels, where n corresponds to the number of rows

# Using a Dictionary of Series

- Let's first construct two Series listing the population and area of the five most populated US states:

```
population_dict = {'California': 39776830,
 'Texas': 28704330,
 'Florida': 21312211,
 'New York': 19862512,
 'Pennsylvania': 12823989}

population = pd.Series(population_dict)
```

```
area_dict = {'California': 423967,
 'Texas': 695662,
 'Florida': 170312,
 'New York': 141297,
 'Pennsylvania': 119282}

area = pd.Series(area_dict)
```

# Using a Dictionary of Series

- Now we can use a dictionary of these two series to construct a DataFrame object:

```
states = pd.DataFrame({'population': population,
 'area': area})

states
```

|              | area   | population |
|--------------|--------|------------|
| California   | 423967 | 39776830   |
| Florida      | 170312 | 21312211   |
| New York     | 141297 | 19862512   |
| Pennsylvania | 119282 | 12823989   |
| Texas        | 695662 | 28704330   |

# DataFrame Attributes

- ▶ The row index labels and column labels can be accessed via the **index** and **columns** attributes:

```
states.index
```

```
Index(['California', 'Florida', 'New York', 'Pennsylvania',
'Texas'], dtype='object')
```

```
states.columns
```

```
Index(['area', 'population'], dtype='object')
```

- ▶ The **values** attribute returns the data contained in the DataFrame as a 2D ndarray

```
states.values
```

```
array([[423967, 39776830],
 [170312, 21312211],
 [141297, 19862512],
 [119282, 12823989],
 [695662, 28704330]], dtype=int64)
```

# DataFrame As Specialized Dictionary

- ▶ Similarly, we can also think of a DataFrame as a specialization of a dictionary
- ▶ Where a dictionary maps a key to a value, a DataFrame maps a column name to a Series of column data
- ▶ For example, asking for the 'area' attribute returns the areas Series object:

```
states['area']
```

|                          |        |
|--------------------------|--------|
| California               | 423967 |
| Florida                  | 170312 |
| New York                 | 141297 |
| Pennsylvania             | 119282 |
| Texas                    | 695662 |
| Name: area, dtype: int64 |        |

- ▶ Notice the potential confusion here: in a 2D NumPy array, `data[0]` will return the first *row*, while in a DataFrame, `data['col0']` will return the first *column*

# DataFrame from a Single Series Object

- ▶ A single-column DataFrame can be constructed from a single Series:

```
pd.DataFrame(population, columns=['population'])
```

|              | population |
|--------------|------------|
| California   | 39776830   |
| Florida      | 21312211   |
| New York     | 19862512   |
| Pennsylvania | 12823989   |
| Texas        | 28704330   |

# DataFrame from a Dictionary of Lists

- ▶ One of the most common ways to create a DataFrame is from a dictionary of equal-length lists or NumPy arrays:

```
data = {'population': [39776830, 28704330, 21312211, 19862512, 12823989],
 'area': [423967, 695662, 170312, 141297, 119282]}
index = ['California', 'Texas', 'Florida', 'New York', 'Pennsylvania']
states = pd.DataFrame(data, index=index)
states
```

|              | area   | population |
|--------------|--------|------------|
| California   | 423967 | 39776830   |
| Texas        | 695662 | 28704330   |
| Florida      | 170312 | 21312211   |
| New York     | 141297 | 19862512   |
| Pennsylvania | 119282 | 12823989   |

- ▶ Note that the columns are placed in sorted order

# DataFrame from a Dictionary of Lists

- ▶ You can specify the sequence of columns, and they will appear in the specified order:

```
data = {'population': [39776830, 28704330, 21312211, 19862512, 12823989],
 'area': [423967, 695662, 170312, 141297, 119282]}
index = ['California', 'Texas', 'Florida', 'New York', 'Pennsylvania']
states = pd.DataFrame(data, index=index,
 columns=['population', 'area'])
states
```

|              | population | area   |
|--------------|------------|--------|
| California   | 39776830   | 423967 |
| Texas        | 28704330   | 695662 |
| Florida      | 21312211   | 170312 |
| New York     | 19862512   | 141297 |
| Pennsylvania | 12823989   | 119282 |

# DataFrame from a Dictionary of Dictionaries

- ▶ DataFrames can also be created by using a nested dictionary of dictionaries
  - ▶ The outer dictionary keys are interpreted as the columns
  - ▶ The keys in the inner dictionaries are unioned and sorted to form the row indices

```
data = {'population': {'California': 39776830,
 'Texas': 28704330,
 'Florida': 21312211,
 'New York': 19862512,
 'Pennsylvania': 12823989},
 'area': {'California': 423967,
 'Texas': 695662,
 'Florida': 170312,
 'New York': 141297,
 'Pennsylvania': 119282}
 }
states = pd.DataFrame(data)
states
```

|              | area   | population |
|--------------|--------|------------|
| California   | 423967 | 39776830   |
| Florida      | 170312 | 21312211   |
| New York     | 141297 | 19862512   |
| Pennsylvania | 119282 | 12823989   |
| Texas        | 695662 | 28704330   |

# DataFrame from a Dictionary of Dictionaries

- If some keys in one of the inner dictionaries are missing, Pandas will fill them in with NaN values:

```
data = {'population': {'California': 39776830,
 'Texas': 28704330,
 'Florida': 21312211,
 'New York': 19862512},
 'area': {'California': 423967,
 'Texas': 695662,
 'New York': 141297,
 'Pennsylvania': 119282}
 }
states = pd.DataFrame(data)
states
```

|              | area     | population |
|--------------|----------|------------|
| California   | 423967.0 | 39776830.0 |
| Florida      | Nan      | 21312211.0 |
| New York     | 141297.0 | 19862512.0 |
| Pennsylvania | 119282.0 | Nan        |
| Texas        | 695662.0 | 28704330.0 |

# DataFrame from a List of Dictionaries

- ▶ Any list of dictionaries can be made into a DataFrame
- ▶ The dictionary keys are interpreted as the columns

```
data = [population_dict, area_dict]
states = pd.DataFrame(data, index=['population', 'area'])
states
```

|            | California | Florida  | New York | Pennsylvania | Texas    |
|------------|------------|----------|----------|--------------|----------|
| population | 39776830   | 21312211 | 19862512 | 12823989     | 28704330 |
| area       | 423967     | 170312   | 141297   | 119282       | 695662   |

- ▶ Of course you can always transpose the result:

```
states.T
```

|              | population | area   |
|--------------|------------|--------|
| California   | 39776830   | 423967 |
| Florida      | 21312211   | 170312 |
| New York     | 19862512   | 141297 |
| Pennsylvania | 12823989   | 119282 |
| Texas        | 28704330   | 695662 |

# DataFrame from a 2D NumPy array

- Given a two-dimensional array of data, we can create a DataFrame with any specified column and index names

```
pd.DataFrame(np.random.rand(3, 2),
 columns=['Column A', 'Column B'],
 index=['a', 'b', 'c'])
```

|   | Column A | Column B |
|---|----------|----------|
| a | 0.188171 | 0.090570 |
| b | 0.308763 | 0.254368 |
| c | 0.630323 | 0.761876 |

- If omitted, an integer index will be used for each:

```
pd.DataFrame(np.random.rand(3, 3))
```

|   | 0        | 1        | 2        |
|---|----------|----------|----------|
| 0 | 0.237744 | 0.876218 | 0.583536 |
| 1 | 0.059770 | 0.177468 | 0.564984 |
| 2 | 0.510760 | 0.789046 | 0.486915 |

# DataFrame from a NumPy Structured Array

- ▶ A Pandas DataFrame operates much like a structured array, and can be created directly from one:

```
dtype = [('name', 'U10'), ('height', float), ('age', int)]
values = [('Arthur', 1.83, 41), ('Lancelot', 1.91, 38), ('Galahad', 1.75, 38)]
a = np.array(values, dtype=dtype)
pd.DataFrame(a)
```

|   | name     | height | age |
|---|----------|--------|-----|
| 0 | Arthur   | 1.83   | 41  |
| 1 | Lancelot | 1.91   | 38  |
| 2 | Galahad  | 1.75   | 38  |

# DataFrame Construction Summary

- The following table shows the possible data inputs to DataFrame constructor:

| Type                             | Description                                                                                                                                |
|----------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| dict of Series                   | Each Series becomes a column. Indexes from each Series are unioned together to form the result's row index if no explicit index is passed. |
| dict of arrays, lists, or tuples | Each sequence becomes a column in the DataFrame.<br>All sequences must be the same length.                                                 |
| dict of dicts                    | Each inner dict becomes a column. Keys are unioned to form the row index as in the “dict of Series” case.                                  |
| list of dicts or Series          | Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become the DataFrame's column labels.                       |
| 2D ndarray                       | A matrix of data, passing optional row and column labels                                                                                   |
| NumPy structured array           | Treated as the “dict of arrays” case                                                                                                       |
| list of lists or tuples          | Treated as the “2D ndarray” case                                                                                                           |
| Another DataFrame                | The DataFrame's indexes are used unless different ones are passed                                                                          |

# Exercise

- ▶ Create the following DataFrame in at least 3 different ways:

|                              | AMZN    | FB      | GOOG    |
|------------------------------|---------|---------|---------|
| <b>Closing price</b>         | 2039.51 | 171.16  | 1197.00 |
| <b>EPS</b>                   | 12.63   | 6.46    | 23.16   |
| <b>Shares Outstanding(M)</b> | 487.74  | 2398.61 | 348.95  |
| <b>Beta</b>                  | 1.61    | 0.66    | 1.40    |
| <b>P/E</b>                   | 157.98  | 25.87   | 51.38   |
| <b>Market Cap(B)</b>         | 972.96  | 482.68  | 829.37  |

# Data Indexing and Selection

- ▶ In NumPy we looked in detail at methods to access, set, and modify values in arrays
- ▶ These included:
  - ▶ Indexing (e.g., `arr[2, 1]`)
  - ▶ Slicing (e.g., `arr[:, 1:5]`)
  - ▶ Masking (e.g., `arr[arr > 0]`)
  - ▶ Fancy indexing (e.g., `arr[0, [1, 5]]`)
  - ▶ And combinations thereof (e.g., `arr[:, [1, 5]]`)
- ▶ Pandas has similar means of accessing and modifying values in Pandas Series and DataFrame objects
- ▶ We'll start with the simple case of the one-dimensional Series object, and then move on to the more complicated two-dimensional DataFrame object

# Selecting Columns

- ▶ A column in a DataFrame can be retrieved as a Series by a dictionary-like notation

```
pop_df['city']
```

```
0 NYC
1 NYC
2 LA
3 LA
4 Chicago
Name: city, dtype: object
```

- ▶ or by attribute:

- ▶ this only works if the index element is a valid Python identifier

```
pop_df.year
```

```
0 2010
1 2015
2 2010
3 2015
4 2015
Name: year, dtype: int64
```

# Selecting Columns

- ▶ We can pass a list of columns to the [] operator in order to select the columns in a particular order:

```
pop_df[['city', 'pop']]
```

|   | city    | pop  |
|---|---------|------|
| 0 | NYC     | 8.19 |
| 1 | NYC     | 8.51 |
| 2 | LA      | 3.80 |
| 3 | LA      | 3.95 |
| 4 | Chicago | 2.71 |

- ▶ Note that rows cannot be selected with the bracket operator [] in a DataFrame
  - ▶ This was a design decision made by the creators in order to avoid ambiguity

# Adding Columns

- ▶ A new column can be added via assignment, as follows:

```
pop_df['state'] = ['NY', 'NY', 'CA', 'CA', 'IL']
pop_df['males%'] = 0.492
pop_df
```

|   | city    | pop  | year | state | males% |
|---|---------|------|------|-------|--------|
| 0 | NYC     | 8.19 | 2010 | NY    | 0.492  |
| 1 | NYC     | 8.51 | 2015 | NY    | 0.492  |
| 2 | LA      | 3.80 | 2010 | CA    | 0.492  |
| 3 | LA      | 3.95 | 2015 | CA    | 0.492  |
| 4 | Chicago | 2.71 | 2015 | IL    | 0.492  |

# Adding Columns

- ▶ A DataFrame structure can be treated as if it were a dictionary of Series objects
- ▶ To insert a column at a specific location, you can use the **insert()** method:

```
pop_df.insert(4, 'females%', 1 - pop_df['males%'])
pop_df
```

|   | city    | pop  | year | state | females% | males% |
|---|---------|------|------|-------|----------|--------|
| 0 | NYC     | 8.19 | 2010 | NY    | 0.508    | 0.492  |
| 1 | NYC     | 8.51 | 2015 | NY    | 0.508    | 0.492  |
| 2 | LA      | 3.80 | 2010 | CA    | 0.508    | 0.492  |
| 3 | LA      | 3.95 | 2015 | CA    | 0.508    | 0.492  |
| 4 | Chicago | 2.71 | 2015 | IL    | 0.508    | 0.492  |

# Alignment

- ▶ DataFrame objects align in a manner similar to Series objects, except that they align on both column and index labels
- ▶ The resulting object is the union of the column and row labels:

```
ore1_df = pd.DataFrame(np.array([[20, 35, 25, 20],
 [11, 28, 32, 29]]),
 columns=['iron', 'magnesium', 'copper', 'silver'])
ore1_df
```

|   | iron | magnesium | copper | silver |
|---|------|-----------|--------|--------|
| 0 | 20   | 35        | 25     | 20     |
| 1 | 11   | 28        | 32     | 29     |

```
ore1_df + ore2_df
```

|   | copper | gold | iron | magnesium | silver |
|---|--------|------|------|-----------|--------|
| 0 | NaN    | NaN  | 34   | 69        | 46     |
| 1 | NaN    | NaN  | 44   | 47        | 52     |

```
ore2_df = pd.DataFrame(np.array([[14, 34, 26, 26],
 [33, 19, 25, 23]]),
 columns=['iron', 'magnesium', 'gold', 'silver'])
ore2_df
```

|   | iron | magnesium | gold | silver |
|---|------|-----------|------|--------|
| 0 | 14   | 34        | 26   | 26     |
| 1 | 33   | 19        | 25   | 23     |

- ▶ When there are no row labels or column labels in common, the value is filled with NaN

# Alignment

- If you combine a DataFrame object and a Series object, the default behavior is to broadcast the Series object across the rows:

```
ore1_df + pd.Series([25, 25, 25, 25],
 index=['iron', 'magnesium', 'copper', 'silver'])
```

|   | iron | magnesium | copper | silver |
|---|------|-----------|--------|--------|
| 0 | 45   | 60        | 50     | 45     |
| 1 | 36   | 53        | 57     | 54     |

# Function Application

- ▶ NumPy ufuncs (element-wise array methods) work fine with pandas objects:

```
np.sqrt(ore1_df)
```

|   | iron     | magnesium | copper   | silver   |
|---|----------|-----------|----------|----------|
| 0 | 4.472136 | 5.916080  | 5.000000 | 4.472136 |
| 1 | 3.316625 | 5.291503  | 5.656854 | 5.385165 |

- ▶ DataFrame's **apply()** method allows you to apply a function on 1D arrays to each column or row:

```
f = lambda x: x.max() - x.min()
ore1_df.apply(f)
```

```
iron 9
magnesium 7
copper 7
silver 9
dtype: int64
```

```
ore1_df.apply(f, axis=1)
```

```
0 15
1 21
dtype: int64
```

## Exercise

- ▶ Consider the following Python dictionary data and Python list labels:

```
data = {'animal': ['cat', 'cat', 'snake', 'dog', 'dog', 'cat', 'snake', 'cat', 'dog', 'dog'],
 'age': [2.5, 3, 0.5, np.nan, 5, 2, 4.5, np.nan, 7, 3],
 'visits': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
 'priority': ['yes', 'yes', 'no', 'yes', 'no', 'no', 'no', 'yes', 'no', 'no']}

labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

- ▶ Create a DataFrame `animals_df` from this dictionary `data` which has the index `labels`
- ▶ Select just the 'animal' and 'age' columns from `animals_df`

# Slicing

- ▶ DataFrame rows can be selected using **slicing**, similarly to NumPy arrays
- ▶ For example, to obtain the first row in the DataFrame:

```
pop_df[0:1]
```

|   | city | pop  | year |
|---|------|------|------|
| 0 | NYC  | 8.19 | 2010 |

- ▶ To obtain all rows starting from index 2:

```
pop_df[2:]
```

|   | city    | pop  | year |
|---|---------|------|------|
| 2 | LA      | 3.80 | 2010 |
| 3 | LA      | 3.95 | 2015 |
| 4 | Chicago | 2.71 | 2015 |

# Slicing

- ▶ Obtain rows at intervals of two, starting from row 0:

```
pop_df[::2]
```

|   | city    | pop  | year |
|---|---------|------|------|
| 0 | NYC     | 8.19 | 2010 |
| 2 | LA      | 3.80 | 2010 |
| 4 | Chicago | 2.71 | 2015 |

- ▶ Reverse the order of rows in DataFrame:

```
pop_df[::-1]
```

|   | city    | pop  | year |
|---|---------|------|------|
| 4 | Chicago | 2.71 | 2015 |
| 3 | LA      | 3.95 | 2015 |
| 2 | LA      | 3.80 | 2010 |
| 1 | NYC     | 8.51 | 2015 |
| 0 | NYC     | 8.19 | 2010 |

# Integer-Based Indexing

- ▶ The `.iloc` attribute supports integer-based positional indexing
- ▶ It accepts the following as inputs:
  - ▶ A single integer, for example, 7
  - ▶ A list or array of integers, for example, [2, 3]
    - ▶ Each integer corresponds to a different axis
  - ▶ A slice object with integers, for example, 1:4
  - ▶ Any combination of the above, for example [5, 2:4] will select row no. 5 and column 2, 3
- ▶ For example, selecting row no. 2:

```
pop_df.iloc[2]
```

```
city LA
pop 3.8
year 2010
Name: 2, dtype: object
```

- ▶ Indexing using a single integer results in a Series object

# Integer-Based Indexing

- ▶ Selecting cell (1, 1):

```
pop_df.iloc[1, 1]
```

8.51

- ▶ Selecting rows 1 and 3, and the first 2 columns:

```
pop_df.iloc[[1, 3], 0:2]
```

|   | city | pop  |
|---|------|------|
| 1 | NYC  | 8.51 |
| 3 | LA   | 3.95 |

- ▶ Indexing using a list of integers results in a DataFrame object

# Integer-Based Indexing

- ▶ The **.iat** attribute can be used for a quick selection of scalar values

```
pop_df.iloc[3, 1]
```

```
3.95
```

```
pop_df.iat[3, 1]
```

```
3.95
```

```
%timeit pop_df.iloc[3, 1]
```

```
9.76 µs ± 637 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
%timeit pop_df.iat[3, 1]
```

```
6.06 µs ± 29.3 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

- ▶ `%timeit` is an IPython magic function, which can be used to time a particular piece of code

# Label Indexing

- ▶ The **.loc** attribute supports pure label-based indexing
- ▶ It accepts the following as valid inputs:
  - ▶ A single label such as ['March'] or ['Dubai']
  - ▶ List or array of labels, for example, ['Dubai', 'UK Brent']
  - ▶ A slice object with labels, for example, 'May':'Aug'
  - ▶ A boolean array
- ▶ For our illustrative dataset, we'll use the average snowy weather temperature data for New York city from the following sites:
  - ▶ <http://www.currentresults.com/Weather/New-York/Places/new-york-city-snowfall-totals-snow-accumulation-averages.php>
  - ▶ <http://www.currentresults.com/Weather/New-York/Places/new-york-city-temperatures-by-month-average.php>

# Label Indexing

## ▶ Create the DataFrame:

```
nyc_snow_avgs_data = {'Months': ['January', 'February', 'March',
 'April', 'November', 'December'],
 'Avg Snow Days': [4.0, 2.7, 1.7, 0.2, 0.2, 2.3],
 'Avg Precip. (cm)': [17.8, 22.4, 9.1, 1.5, 0.8, 12.2],
 'Avg Low Temp. (C)': [-3, -2, 2, 7, 5, 0] }
```

```
nyc_snow_avgs = pd.DataFrame(nyc_snow_avgs_data,
 index = nyc_snow_avgs_data['Months'],
 columns = ['Avg Snow Days', 'Avg Precip. (cm)',
 'Avg Low Temp. (C)'])
nyc_snow_avgs
```

|          | Avg Snow Days | Avg Precip. (cm) | Avg Low Temp. (C) |
|----------|---------------|------------------|-------------------|
| January  | 4.0           | 17.8             | -3                |
| February | 2.7           | 22.4             | -2                |
| March    | 1.7           | 9.1              | 2                 |
| April    | 0.2           | 1.5              | 7                 |
| November | 0.2           | 0.8              | 5                 |
| December | 2.3           | 12.2             | 0                 |

# Label Indexing

- ▶ Using a single label:

```
nyc_snow_avgs.loc['January']
```

```
Avg Snow Days 4.0
Avg Precip. (cm) 17.8
Avg Low Temp. (C) -3.0
Name: January, dtype: float64
```

- ▶ Using a list of labels:

```
nyc_snow_avgs.loc[['January', 'April']]
```

|         | Avg Snow Days | Avg Precip. (cm) | Avg Low Temp. (C) |
|---------|---------------|------------------|-------------------|
| January | 4.0           | 17.8             | -3                |
| April   | 0.2           | 1.5              | 7                 |

# Label Indexing

- ▶ Using a label range:

```
nyc_snow_avgs.loc['January': 'March']
```

|          | Avg SnowDays | Avg Precip. (cm) | Avg Low Temp. (C) |
|----------|--------------|------------------|-------------------|
| January  | 4.0          | 17.8             | -3                |
| February | 2.7          | 22.4             | -2                |
| March    | 1.7          | 9.1              | 2                 |

- ▶ Note that while using the .loc and .iloc operators, the row index must always be specified first
  - ▶ In contrast to the [] operator, where columns can be selected directly
- ▶ Hence, we get an error if we do the following:

```
nyc_snow_avgs.loc['Avg Snow Days']
```

KeyError

Traceback (most recent call last)

# Label Indexing

- ▶ The correct way to do this is to specifically select all rows by using colon (:) operator:

```
nyc_snow_avgs.loc[:, 'Avg Snow Days']
```

```
January 4.0
February 2.7
March 1.7
April 0.2
November 0.2
December 2.3
Name: Avg Snow Days, dtype: float64
```

- ▶ You can select a specific cell value using a row label index and a column label index
- ▶ The following example shows the average number of snow days in March:

```
nyc_snow_avgs.loc['March', 'Avg Snow Days']
```

```
1.7
```

```
nyc_snow_avgs.loc['March']['Avg Snow Days']
```

```
1.7
```

# Boolean Indexing

- ▶ You can also filter rows using a boolean array (similarly to NumPy arrays)
- ▶ For example, we can select which months have less than one snow day on average:

```
nyc_snow_avgs['Avg Snow Days'] < 1
```

```
January False
February False
March False
April True
November True
December False
Name: Avg Snow Days, dtype: bool
```

```
nyc_snow_avgs[nyc_snow_avgs['Avg Snow Days'] < 1]
```

|          | Avg Snow Days | Avg Precip. (cm) | Avg Low Temp. (C) |
|----------|---------------|------------------|-------------------|
| April    | 0.2           | 1.5              | 7                 |
| November | 0.2           | 0.8              | 5                 |

# Boolean Indexing

- ▶ As for NumPy arrays, you can use the logical operators | (OR), & (AND), ~ (NOT) in boolean indexing
- ▶ The following example selects which months have more than 2 snow days on average and their average low temperature is less than 0:

```
nyc_snow_avgs[(nyc_snow_avgs['Avg Snow Days'] > 2) &
 (nyc_snow_avgs['Avg Low Temp. (C)'] < 0)]
```

|          | Avg Snow Days | Avg Precip. (cm) | Avg Low Temp. (C) |
|----------|---------------|------------------|-------------------|
| January  | 4.0           | 17.8             | -3                |
| February | 2.7           | 22.4             | -2                |

## Exercise

- ▶ Implement the following operations on the DataFrame `animals_df`:
  - ▶ Return the first 3 rows of the DataFrame
  - ▶ Select the number of visits of the animal in row 'd'
  - ▶ Select the data in rows ['b', 'e'] and in columns ['animal', 'age']
  - ▶ Select the data in rows [3, 4, 8] and in columns ['animal', 'age']
  - ▶ Select the rows where the number of visits is greater than 2
  - ▶ Select the rows where the animal is a cat and the age is less than 3
  - ▶ Select the rows where the age is missing, i.e. is `NaN`
  - ▶ Change the age in row 'f' to 1.5

# Sorting

- ▶ Sorting a data set by some criterion is another important built-in operation
- ▶ To sort lexicographically by row or column index, use the **sort\_index()** method, which returns a new, sorted object
- ▶ For example, sorting a Series object by its index:

```
ser = pd.Series(range(4), index=['d', 'b', 'a', 'c'])
ser.sort_index()
```

```
a 2
b 1
c 3
d 0
dtype: int64
```

- ▶ To sort a Series by its values, user the **sort\_values()** method:

```
ser2 = pd.Series([4, 7, -3, 2])
ser2.sort_values()
```

```
2 -3
3 2
0 4
1 7
dtype: int64
```

# Sorting

- With a DataFrame, you can sort by index on either axis:

```
df = pd.DataFrame(np.arange(8).reshape(2, 4), index=['three', 'one'],
 columns=['d', 'b', 'a', 'c'])
df
```

|       | d | b | a | c |
|-------|---|---|---|---|
| three | 0 | 1 | 2 | 3 |
| one   | 4 | 5 | 6 | 7 |

```
df.sort_index()
```

|       | d | b | a | c |
|-------|---|---|---|---|
| one   | 4 | 5 | 6 | 7 |
| three | 0 | 1 | 2 | 3 |

```
df.sort_index(axis=1)
```

|       | a | b | c | d |
|-------|---|---|---|---|
| three | 2 | 1 | 3 | 0 |
| one   | 6 | 5 | 7 | 4 |

# Sorting

- To sort a DataFrame by the values in the columns, use **sort\_values()** with the column indexes:

```
df = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
df
```

|   | a | b  |
|---|---|----|
| 0 | 0 | 4  |
| 1 | 1 | 7  |
| 2 | 0 | -3 |
| 3 | 1 | 2  |

```
df.sort_values('b')
```

|   | a | b  |
|---|---|----|
| 2 | 0 | -3 |
| 3 | 1 | 2  |
| 0 | 0 | 4  |
| 1 | 1 | 7  |

# Sorting

- ▶ To sort by multiple columns, pass a list of names:

```
df.sort_values(['a', 'b'])
```

|   | a | b  |
|---|---|----|
| 2 | 0 | -3 |
| 0 | 0 | 4  |
| 3 | 1 | 2  |
| 1 | 1 | 7  |

- ▶ The **ascending** argument can be used to change the sorting order (default is ascending)

```
df.sort_values('b', ascending=False)
```

|   | a | b  |
|---|---|----|
| 1 | 1 | 7  |
| 0 | 0 | 4  |
| 3 | 1 | 2  |
| 2 | 0 | -3 |

# Sorting

- If you want to sort each column by a different order, you can pass a list of booleans to the ascending argument:

```
df.sort_values(['a', 'b'], ascending=[False, True])
```

|   | a | b  |
|---|---|----|
| 3 | 1 | 2  |
| 1 | 1 | 7  |
| 2 | 0 | -3 |
| 0 | 0 | 4  |

- You can use the **inplace** argument to sort the data in place:

```
df.sort_values('b', inplace=True)
df
```

|   | a | b  |
|---|---|----|
| 2 | 0 | -3 |
| 3 | 1 | 2  |
| 0 | 0 | 4  |
| 1 | 1 | 7  |

# Computing Descriptive Statistics

- ▶ pandas objects are equipped with a set of common statistical methods
- ▶ Most of these methods extract a single value (like the sum or mean) from a Series, or a series of values from the rows or columns of a DataFrame
- ▶ Compared with the equivalent methods of vanilla NumPy arrays, they are all built from the ground up to exclude missing data

# Computing Descriptive Statistics

- ▶ A list of summary statistics and related methods:

| Method         | Description                                                                                 |
|----------------|---------------------------------------------------------------------------------------------|
| count          | Number of non-NA values                                                                     |
| min, max       | Compute minimum and maximum values                                                          |
| argmin, argmax | Compute index locations (integers) at which minimum or maximum value obtained, respectively |
| idxmin, idxmax | Compute index values at which minimum or maximum value obtained, respectively               |
| sum            | Sum of values                                                                               |
| mean           | Mean of values                                                                              |
| mad            | Mean absolute deviation from mean value                                                     |
| median         | Arithmetic median (50% quantile) of values                                                  |
| quantile       | Compute sample quantile ranging from 0 to 1                                                 |
| var            | Sample variance of values                                                                   |
| std            | Sample standard deviation of values                                                         |
| cumsum         | Cumulative sum of values                                                                    |
| describe       | Compute set of summary statistics for Series or each DataFrame column                       |

# Computing Descriptive Statistics

- ▶ Consider a small DataFrame:

```
df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5], [np.nan, np.nan], [0.75, -1.3]],
 index=['a', 'b', 'c', 'd'],
 columns=['one', 'two'])
df
```

|   | one  | two  |
|---|------|------|
| a | 1.40 | NaN  |
| b | 7.10 | -4.5 |
| c | NaN  | NaN  |
| d | 0.75 | -1.3 |

- ▶ Calling DataFrame's **sum()** method returns a Series containing column sums:

```
df.sum()
```

```
one 9.25
two -5.80
dtype: float64
```

# Computing Descriptive Statistics

- ▶ Passing **axis=1** sums over the rows instead:

```
df.sum(axis=1)
```

```
a 1.40
b 2.60
c 0.00
d -0.55
dtype: float64
```

- ▶ You can take into account NA values by using the **skipna** option:

```
df.sum(axis=1, skipna=False)
```

```
a NaN
b 2.60
c NaN
d -0.55
dtype: float64
```

# Computing Descriptive Statistics

- ▶ Some methods, like **idxmin** and **idxmax**, return indirect statistics like the index value where the minimum or maximum values are attained:

```
df.idxmax()
```

```
one b
two d
dtype: object
```

- ▶ Other methods are *accumulations*:

```
df.cumsum()
```

|   | one  | two  |
|---|------|------|
| a | 1.40 | NaN  |
| b | 8.50 | -4.5 |
| c | NaN  | NaN  |
| d | 9.25 | -5.8 |

# Computing Descriptive Statistics

- ▶ The method **describe()** produces multiple summary statistics in one shot:

```
df.describe()
```

|              | one      | two       |
|--------------|----------|-----------|
| <b>count</b> | 3.000000 | 2.000000  |
| <b>mean</b>  | 3.083333 | -2.900000 |
| <b>std</b>   | 3.493685 | 2.262742  |
| <b>min</b>   | 0.750000 | -4.500000 |
| <b>25%</b>   | 1.075000 | -3.700000 |
| <b>50%</b>   | 1.400000 | -2.900000 |
| <b>75%</b>   | 4.250000 | -2.100000 |
| <b>max</b>   | 7.100000 | -1.300000 |

- ▶ On non-numeric data, describe produces alternate summary statistics:

```
ser = pd.Series(['a', 'a', 'b', 'c'] * 4)
ser.describe()
```

```
count 16
unique 3
top a
freq 8
dtype: object
```

# Correlation and Covariance

- ▶ Some summary statistics, like correlation and covariance, are computed from pairs of arguments
- ▶ Let's consider DataFrames of stock prices obtained from the web
- ▶ We first need to install the **pandas\_datareader** module
  - ▶ Functions from pandas\_datareader.data and pandas\_datareader.wb extract data from various Internet sources into a pandas DataFrame
- ▶ Open Anaconda command prompt and type:

```
pip install pandas-datareader
```

# Correlation and Covariance

- ▶ The following code downloads all the stock data between 1/1/2010 and 1/1/2018 of the specified tickers:

```
import pandas_datareader.data as web
start = '1/1/2015'
end = '1/1/2018'

stocks_data = {}
for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']:
 stocks_data[ticker] = web.DataReader(ticker, 'iex', start, end)
```

```
stocks_data
```

|            | open     | high     | low      | close    | volume   |
|------------|----------|----------|----------|----------|----------|
| 'AAPL':    |          |          |          |          |          |
| date       |          |          |          |          |          |
| 2015-01-02 | 104.6128 | 104.6597 | 100.8186 | 102.6781 | 53204626 |
| 2015-01-05 | 101.7014 | 102.0395 | 98.9966  | 99.7855  | 64285491 |
| 2015-01-06 | 100.0579 | 100.8937 | 98.2641  | 99.7949  | 65797116 |
| 2015-01-07 | 100.6777 | 101.6169 | 100.2034 | 101.1942 | 40105934 |
| 2015-01-08 | 102.5842 | 105.3265 | 102.0864 | 105.0823 | 59364547 |
| 2015-01-09 | 105.8149 | 106.3596 | 103.5046 | 105.1950 | 53699527 |

- ▶ `stocks_data` is a dictionary containing a DataFrame for every specified ticker
- ▶ The index of the DataFrame is the date and its columns are open, high, low, close and volume

# Correlation and Covariance

- ▶ We now create a DataFrame of the prices of all the tickers, indexed by the date:

```
prices_df = pd.DataFrame({tic: data['close']
 for tic, data in stocks_data.items()})
prices_df.head()
```

|            | AAPL     | GOOG   | IBM      | MSFT    |
|------------|----------|--------|----------|---------|
| date       |          |        |          |         |
| 2015-01-02 | 102.6781 | 524.81 | 142.5291 | 42.9486 |
| 2015-01-05 | 99.7855  | 513.87 | 140.2865 | 42.5490 |
| 2015-01-06 | 99.7949  | 501.96 | 137.2610 | 41.9290 |
| 2015-01-07 | 101.1942 | 501.10 | 136.3640 | 42.4618 |
| 2015-01-08 | 105.0823 | 502.68 | 139.3278 | 43.7109 |

- ▶ The method **head(n)** returns the first n rows of the DataFrame (the default is n = 5)

# Correlation and Covariance

- ▶ We now compute percent changes of the prices:

```
price_change = prices_df.pct_change()
price_change.head()
```

|            | AAPL      | GOOG      | IBM       | MSFT      |
|------------|-----------|-----------|-----------|-----------|
| date       |           |           |           |           |
| 2015-01-02 | NaN       | NaN       | NaN       | NaN       |
| 2015-01-05 | -0.028172 | -0.020846 | -0.015734 | -0.009304 |
| 2015-01-06 | 0.000094  | -0.023177 | -0.021567 | -0.014571 |
| 2015-01-07 | 0.014022  | -0.001713 | -0.006535 | 0.012707  |
| 2015-01-08 | 0.038422  | 0.003153  | 0.021734  | 0.029417  |

# Correlation and Covariance

- ▶ The **corr()** method of Series computes the correlation of the overlapping, non-NA, aligned-by-index values in two Series
- ▶ By default, it uses Pearson correlation measure

$$r = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$$

- ▶ where n is the sample size,  $x_i$  and  $y_i$  are the sample points and  $\bar{x}$  and  $\bar{y}$  are the sample means
- ▶ For example, the correlation between the price changes of Microsoft and Google stocks is:

```
price_change.MSFT.corr(price_change.GOOG)
```

```
0.5872899839300288
```

# Correlation and Covariance

- ▶ The **cov()** method of Series computes the covariance of the overlapping, non-NA, aligned-by-index values in two Series, normalized by n-1

$$\text{cov}(X, Y) = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})$$

- ▶ For example, the covariance between the price changes of Microsoft and Google stocks is:

```
price_change.MSFT.cov(price_change.GOOG)
```

```
0.00011810635860666633
```

# Correlation and Covariance

- ▶ DataFrame's **corr()** and **cov()** methods return a full correlation or covariance matrix as a DataFrame, respectively:

```
price_change.corr()
```

|      | AAPL     | GOOG     | IBM      | MSFT     |
|------|----------|----------|----------|----------|
| AAPL | 1.000000 | 0.419992 | 0.330654 | 0.494976 |
| GOOG | 0.419992 | 1.000000 | 0.337053 | 0.587290 |
| IBM  | 0.330654 | 0.337053 | 1.000000 | 0.416134 |
| MSFT | 0.494976 | 0.587290 | 0.416134 | 1.000000 |

```
price_change.cov()
```

|      | AAPL     | GOOG     | IBM      | MSFT     |
|------|----------|----------|----------|----------|
| AAPL | 0.000211 | 0.000086 | 0.000057 | 0.000102 |
| GOOG | 0.000086 | 0.000200 | 0.000057 | 0.000118 |
| IBM  | 0.000057 | 0.000057 | 0.000144 | 0.000071 |
| MSFT | 0.000102 | 0.000118 | 0.000071 | 0.000202 |

# Correlation and Covariance

- ▶ Using DataFrame's **corrwith()** method, you can compute pairwise correlations between a DataFrame's columns or rows with another Series or DataFrame
- ▶ For example, to compute the correlation between IBM stock's price change and the other stocks price changes:

```
price_change.corrwith(price_change.IBM)
```

```
AAPL 0.330654
GOOG 0.337053
IBM 1.000000
MSFT 0.416134
dtype: float64
```

- ▶ Passing a DataFrame computes the correlations of matching column names

## Exercise

---

- ▶ Calculate the sum of all visits (the total number of visits) in aminals\_df
- ▶ Count the number of cats in the DataFrame
- ▶ Calculate the mean of the ages of all the animals with more than 2 visits
- ▶ Compute the correlation and covariance between the age and visits columns

# Reading and Writing Text Files

- ▶ pandas features a number of functions for reading tabular data as a DataFrame:

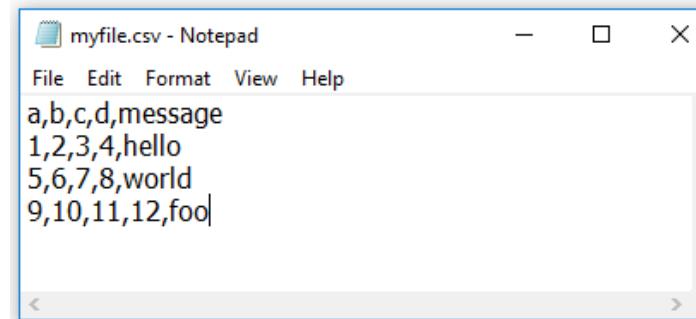
| Function   | Description                                                                                     |
|------------|-------------------------------------------------------------------------------------------------|
| read_csv   | Load delimited data from a file, URL, or file-like object. Use comma as default delimiter.      |
| read_table | Load delimited data from a file, URL, or file-like object. Use tab ('\t') as default delimiter. |
| read_fwf   | Read data in fixed-width column format (that is, no delimiters)                                 |

- ▶ The options for these functions fall into a few categories:
  - ▶ **Indexing:** can treat one or more columns as the returned DataFrame, and whether to get column names from the file, the user, or not at all
  - ▶ **Type inference and data conversion:** this includes user-defined value conversions and custom list of missing value markers
  - ▶ **Datetime parsing:** including combining date and time fields spread over multiple columns
  - ▶ **Iterating:** support for iterating over chunks of very large files
  - ▶ **Unclean data issues:** skipping rows or a footer, comments

# Reading Text Files

- ▶ Let's start with a small comma-separated (CSV) text file
- ▶ Create myfile.csv in your Notebook folder, open it with a text editor and type:

```
a,b,c,d,message
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```



- ▶ Since this is comma-delimited, we can use **read\_csv()** to read it into a DataFrame:

```
df = pd.read_csv('myfile.csv')
df
```

|   | a | b  | c  | d  | message |
|---|---|----|----|----|---------|
| 0 | 1 | 2  | 3  | 4  | hello   |
| 1 | 5 | 6  | 7  | 8  | world   |
| 2 | 9 | 10 | 11 | 12 | foo     |

# Reading Text Files

- ▶ A file will not always have a header row. Consider myfile2.csv:

```
!type myfile2.csv
```

```
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

- ▶ !type executes the type shell command that prints the file contents (same as cat in Linux)
- ▶ To read this in, you can allow pandas to assign default column names, or you can specify names yourself:

```
pd.read_csv('myfile2.csv', header=None)
```

|   | 0 | 1  | 2  | 3  | 4     |
|---|---|----|----|----|-------|
| 0 | 1 | 2  | 3  | 4  | hello |
| 1 | 5 | 6  | 7  | 8  | world |
| 2 | 9 | 10 | 11 | 12 | foo   |

```
pd.read_csv('myfile2.csv', names=['a', 'b', 'c', 'd', 'message'])
```

|   | a | b  | c  | d  | message |
|---|---|----|----|----|---------|
| 0 | 1 | 2  | 3  | 4  | hello   |
| 1 | 5 | 6  | 7  | 8  | world   |
| 2 | 9 | 10 | 11 | 12 | foo     |

# Reading Text Files

- To set the index of the DataFrame to one of the columns, you can either use the **index\_col** argument of `read_csv()`:

```
pd.read_csv('myfile.csv', index_col='message')
```

|         | a | b  | c  | d  |
|---------|---|----|----|----|
| message |   |    |    |    |
| hello   | 1 | 2  | 3  | 4  |
| world   | 5 | 6  | 7  | 8  |
| foo     | 9 | 10 | 11 | 12 |

- Or you can call the **set\_index()** method of the DataFrame:

```
df = pd.read_csv('myfile.csv')
df = df.set_index('message')
df
```

|         | a | b  | c  | d  |
|---------|---|----|----|----|
| message |   |    |    |    |
| hello   | 1 | 2  | 3  | 4  |
| world   | 5 | 6  | 7  | 8  |
| foo     | 9 | 10 | 11 | 12 |

# Reading Text Files

- ▶ If the fields of each row are not separated by comma, you can use the **sep** parameter to specify a character sequence or a regular expression to use to split the fields
  - ▶ You can use either `read_csv()` or `read_table()` in this case
  - ▶ `read_table()` is `read_csv()` with `sep=','` replaced by `sep='\t'`
- ▶ For example, consider the following text file:

| a | b  | c  | d  | message |
|---|----|----|----|---------|
| 1 | 2  | 3  | 4  | hello   |
| 5 | 6  | 7  | 8  | world   |
| 9 | 10 | 11 | 12 | foo     |

- ▶ The fields in each row are separated by a space, thus we use `sep=' '` to read it:

```
pd.read_table('myfile3.txt', sep=' ')
```

|   | a | b  | c  | d  | message |
|---|---|----|----|----|---------|
| 0 | 1 | 2  | 3  | 4  | hello   |
| 1 | 5 | 6  | 7  | 8  | world   |
| 2 | 9 | 10 | 11 | 12 | foo     |

# Reading Text Files

- ▶ In some cases, a table might not have a fixed delimiter
- ▶ For example, consider this text file:

|     | A         | B         | C         |
|-----|-----------|-----------|-----------|
| aaa | -0.264438 | -1.026059 | -0.619500 |
| bbb | 0.927272  | 0.302904  | -0.032399 |
| ccc | -0.264273 | -0.386314 | -0.217601 |
| ddd | -0.871858 | -0.348382 | 1.100491  |

- ▶ In this case fields are separated by a variable amount of whitespace
- ▶ This can be expressed by the regular expression \s+

```
pd.read_table('myfile4.txt', sep='\s+')
```

|     | A         | B         | C         |
|-----|-----------|-----------|-----------|
| aaa | -0.264438 | -1.026059 | -0.619500 |
| bbb | 0.927272  | 0.302904  | -0.032399 |
| ccc | -0.264273 | -0.386314 | -0.217601 |
| ddd | -0.871858 | -0.348382 | 1.100491  |

Because there was one fewer column name than the number of data rows, `read_table()` infers that the first column should be the DataFrame's index

# Reading Text Files

- ▶ The parser functions have many additional arguments to help you handle the wide variety of exception file formats that might occur:

| Argument    | Description                                                                                                                            |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------|
| path        | String indicating filesystem location, URL, or file-like object                                                                        |
| sep         | Character sequence or regular expression to use to split fields in each row                                                            |
| header      | Row number to use as column names. Defaults to 0 (first row), but should be None if there is no header row.                            |
| names       | List of column names for result, combine with header=None                                                                              |
| index_col   | Column numbers or names to use as the row index in the result. Can be a single name/number or a list of them for a hierarchical index. |
| skiprows    | Number of rows at beginning of file to ignore or list of row numbers (starting from 0) to skip                                         |
| skip_footer | Number of lines to ignore at end of file                                                                                               |
| na_values   | Sequence of values to replace with NA                                                                                                  |
| comment     | Character or characters to split comments off the end of lines                                                                         |

# Reading Text Files

| Argument    | Description                                                                                                                                                       |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| converters  | Dict containing column number or name mapping to functions. For example {'foo': f} would apply the function f to all values in the 'foo' column.                  |
| nrows       | Number of rows to read from beginning of file                                                                                                                     |
| chunksize   | For iteration, size of file chunks                                                                                                                                |
| encoding    | Text encoding for unicode. For example 'utf-8' for UTF-8 encoded text.                                                                                            |
| squeeze     | If the parsed data only contains one column return a Series                                                                                                       |
| thousands   | Separator for thousands, e.g. ',' or '.'                                                                                                                          |
| parse_dates | Attempt to parse data to datetime; False by default. If True, will attempt to parse all columns. Otherwise can specify a list of column numbers or name to parse. |
| dayfirst    | When parsing potentially ambiguous dates, treat as international format (e.g. 7/6/2012 -> June 7, 2012). Default False.                                           |
| date_parser | Function to use to parse dates.                                                                                                                                   |

# Reading Text Files

- ▶ For example, you can skip the first, third, and fourth rows of a file with skiprows:

```
!type myfile5.csv
```

```
hey!
a,b,c,d,message
just wanted to make things more difficult for you
who reads CSV files with computers, anyway?
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

```
pd.read_csv('myfile5.csv', skiprows=[0, 2, 3])
```

|   | a | b  | c  | d  | message |
|---|---|----|----|----|---------|
| 0 | 1 | 2  | 3  | 4  | hello   |
| 1 | 5 | 6  | 7  | 8  | world   |
| 2 | 9 | 10 | 11 | 12 | foo     |

# Reading Text Files

- ▶ Handling missing values is an important part of the file parsing process
- ▶ Missing data is usually either not present (empty string) or marked by some **sentinel**
- ▶ By default, pandas recognizes as NaN a set of commonly occurring sentinels, such as NA, nan, n/a, -1.#IND, null, etc.

```
!type myfile6.csv
```

```
something,a,b,c,d,message
one,1,2,3,4,NA
two,5,,6,,8,world
three,9,null,11,12,foo
```

```
pd.read_csv('myfile6.csv')
```

|   | something | a | b   | c    | d  | message |
|---|-----------|---|-----|------|----|---------|
| 0 | one       | 1 | 2.0 | 3.0  | 4  | NaN     |
| 1 | two       | 5 | 6.0 | NaN  | 8  | world   |
| 2 | three     | 9 | NaN | 11.0 | 12 | foo     |

# Reading Text Files

- ▶ The **na\_values** option can take either a list of strings to consider missing values:

```
pd.read_csv('myfile6.csv', na_values=['foo'])
```

|   | something | a | b   | c    | d  | message |
|---|-----------|---|-----|------|----|---------|
| 0 | one       | 1 | 2.0 | 3.0  | 4  | NaN     |
| 1 | two       | 5 | 6.0 | NaN  | 8  | world   |
| 2 | three     | 9 | NaN | 11.0 | 12 | NaN     |

## Exercise

- ▶ Download the file euro\_winners.csv from <https://goo.gl/KnFPfC>
- ▶ This dataset, obtained from Wikipedia, contains data for the finals of the European club championship since its inception in 1955
  - ▶ [https://en.wikipedia.org/wiki/List\\_of\\_European\\_Cup\\_and\\_UEFA\\_Champions\\_League\\_finals](https://en.wikipedia.org/wiki/List_of_European_Cup_and_UEFA_Champions_League_finals)
- ▶ Read the table into a DataFrame
- ▶ Define the season column as the index
- ▶ Display the last 5 rows of the table
- ▶ Display the name of the team who won the championship at season 2000–01 (note the dash in the middle, typed by Alt+0150)
- ▶ Display the three games with the highest number of attendees

# Writing Data To Text Format

- ▶ Data can also be exported to delimited format
- ▶ Let's consider one of the CSV files read above, and change one of the table cells:

```
df = pd.read_csv('myfile6.csv')
df.loc[0, 'message'] = 'test'
df
```

|   | something | a | b   | c    | d  | message |
|---|-----------|---|-----|------|----|---------|
| 0 | one       | 1 | 2.0 | 3.0  | 4  | test    |
| 1 | two       | 5 | 6.0 | NaN  | 8  | world   |
| 2 | three     | 9 | NaN | 11.0 | 12 | foo     |

- ▶ The DataFrame's `to_csv()` method writes the data out to a comma-separated file:

```
df.to_csv('out.csv')
```

```
!type out.csv
```

```
,something,a,b,c,d,message
0,one,1,2.0,3.0,4,test
1,two,5,6.0,,8,world
2,three,9,,11.0,12,foo
```

# Writing Data To Text Format

- ▶ Other delimiters can be used, of course:

```
df.to_csv('out.csv', sep='|')
!type out.csv
```

```
|something|a|b|c|d|message
0|one|1|2.0|3.0|4|test
1|two|5|6.0||8|world
2|three|9||11.0|12|foo
```

- ▶ Missing values appear as empty strings in the output
- ▶ You might want to denote them by some other sentinel value:

```
df.to_csv('out.csv', na_rep='NULL')
!type out.csv
```

```
,something,a,b,c,d,message
0,one,1,2.0,3.0,4,test
1,two,5,6.0,NULL,8,world
2,three,9,NULL,11.0,12,foo
```

# Writing Data To Text Format

- ▶ With no other options specified, both the row and column labels are written
- ▶ Both of these can be disabled:

```
df.to_csv('out.csv', index=False, header=False)
!type out.csv
```

```
one,1,2.0,3.0,4,test
two,5,6.0,,8,world
three,,9,,11.0,12,foo
```

- ▶ You can also write only a subset of the columns, and in an order of your choosing:

```
df.to_csv('out.csv', index=False, columns=['a', 'b', 'c'])
!type out.csv
```

```
a,b,c
1,2.0,3.0
5,6.0,
9,,11.0
```

# Reading Text Files In Pieces

- ▶ When processing very large files, you may only want to read in a small piece of a file or iterate through smaller chunks of the file
- ▶ Let's create a CSV file with 10,000 rows containing random numbers in four columns:

```
df = pd.DataFrame(np.random.rand(10000, 4),
 columns=['one', 'two', 'three', 'four'])
df.to_csv('myfile7.csv', index=False)
```

- ▶ If you want to only read out a small number of rows, specify that with **nrows**:

```
pd.read_csv('myfile7.csv', nrows=5)
```

|   | one      | two      | three    | four     |
|---|----------|----------|----------|----------|
| 0 | 0.763827 | 0.784181 | 0.141241 | 0.759184 |
| 1 | 0.495430 | 0.864463 | 0.467473 | 0.957617 |
| 2 | 0.892346 | 0.322783 | 0.591786 | 0.011954 |
| 3 | 0.209003 | 0.520671 | 0.209522 | 0.474396 |
| 4 | 0.464871 | 0.802502 | 0.387418 | 0.330254 |

# Reading Text Files In Pieces

- ▶ To read out a file in pieces, specify a **chunksize** as a number of rows:

```
reader = pd.read_csv('myfile7.csv', chunksize=1000)
reader
```

```
<pandas.io.parsers.TextFileReader at 0x25db1899588>
```

- ▶ The TextFileReader object returned by `read_csv` is an iterator that allows you to iterate over the parts of the file according to the `chunksize`
- ▶ The following code shows the average of values of the first column in each chunk:

```
for chunk in reader:
 print(chunk['one'].mean())
```

```
0.503411622981677
0.5031108600109644
0.5086342971877635
0.5041450364490075
0.506953671321647
0.5097013179565795
0.49392377946905364
0.4947612599954584
0.4887523295903048
0.4915130217928475
```

# Reading Text Files In Pieces

- TextFileReader also has a method `get_chunk()` that enables you to read chunks of arbitrary size:

```
reader = pd.read_csv('myfile7.csv', chunksize=1000)
reader.get_chunk(5)
```

|   | one      | two      | three    | four     |
|---|----------|----------|----------|----------|
| 0 | 0.763827 | 0.784181 | 0.141241 | 0.759184 |
| 1 | 0.495430 | 0.864463 | 0.467473 | 0.957617 |
| 2 | 0.892346 | 0.322783 | 0.591786 | 0.011954 |
| 3 | 0.209003 | 0.520671 | 0.209522 | 0.474396 |
| 4 | 0.464871 | 0.802502 | 0.387418 | 0.330254 |

```
reader.get_chunk(5)
```

|   | one      | two      | three    | four     |
|---|----------|----------|----------|----------|
| 5 | 0.973990 | 0.783628 | 0.647234 | 0.985077 |
| 6 | 0.337996 | 0.616457 | 0.762881 | 0.742263 |
| 7 | 0.771908 | 0.879579 | 0.300900 | 0.246773 |
| 8 | 0.658600 | 0.622567 | 0.496328 | 0.436382 |
| 9 | 0.294799 | 0.735844 | 0.756918 | 0.258002 |

- ▶ JSON (JavaScript Object Notation) has become one of the standard formats for sending data between applications
- ▶ It is a much more flexible data format than a tabular text form like CSV
  - ▶ In fact, the Jupyter Notebooks themselves (.ipynb files) are stored in JSON format
- ▶ JSON types include objects (dicts), arrays (lists), strings, numbers, booleans, and nulls
- ▶ All of the keys in an object must be strings

```
{
 "Time" : "2014-02-12 14:20:05",
 "Latitude" : 37.33233141,
 "Longitude" : -122.0312186,
 "Count" : 101,
 "Comments" : "Bad Data. SNOW DAY!!",
 "Luma" : 0,
 "Habitat" : "Back yard, grass",
 "Types" : [
 "5",
 "6"
],
 "Address" : {
 "Street" : "2522 West Georgia Road",
 "City" : "Piedmont",
 "State" : "South Carolina",
 "Country" : "United States",
 }
}
```

- ▶ Here is an example for defining a string in a JSON format:

```
json_str = """
{"name": "Mark",
 "age": 35,
 "places_lived": ["US", "Israel", "Spain"],
 "wife": null,
 "siblings": [{"name": "Scott", "age": 25, "pet": "Zuko"},
 {"name": "Katie", "age": 33, "pet": "Kitty"}]}
"""
```

- ▶ JSON is very nearly valid Python code with the exception of its null value null (Python uses None) and some other nuances (such as disallowing trailing commas at the end of lists)
- ▶ There are several Python libraries for reading and writing JSON data
- ▶ We'll use the json module which is part of the Python standard library

- ▶ **json.loads()** converts a JSON string into a Python object (a dictionary or a list):

```
import json

obj = json.loads(json_str)
obj

{'age': 35,
 'name': 'Mark',
 'places_lived': ['US', 'Israel', 'Spain'],
 'siblings': [{`age': 25, 'name': 'Scott', 'pet': 'Zuko'},
 {'age': 33, 'name': 'Katie', 'pet': 'Kitty'}],
 'wife': None}
```

- ▶ **json.dumps()** converts a Python object back to JSON:

```
obj['places_lived'].append('Germany')
json_str2 = json.dumps(obj)
json_str2

'{"name": "Mark", "age": 35, "places_lived": ["US", "Israe
l", "Spain", "Germany"], "wife": null, "siblings": [{"name":
"Scott", "age": 25, "pet": "Zuko"}, {"name": "Katie", "age":
33, "pet": "Kitty"}]}'
```

- ▶ How you convert a JSON object or list of objects to a DataFrame will be up to you
- ▶ For example, you can pass a list of JSON objects (list of dictionaries) to the DataFrame constructor and select a subset of the data fields:

```
siblings_df = pd.DataFrame(obj['siblings'], columns=['name', 'age'])
siblings_df
```

|   | name  | age |
|---|-------|-----|
| 0 | Scott | 25  |
| 1 | Katie | 33  |

- ▶ There is also a fast native JSON export (`to_json`) and decoding (`read_json`) to pandas
- ▶ For example:

```
df = pd.DataFrame([['a', 'b'], ['c', 'd']],
 index=['Row1', 'Row2'],
 columns=['Col1', 'Col2'])
```

```
df.to_json('myfile8.json')
!type myfile8.json
```

```
{"Col1":{"Row1":"a", "Row2":"c"}, "Col2":{"Row1":"b", "Row2":"d"}}
```

```
pd.read_json('myfile8.json')
```

|      | Col1 | Col2 |
|------|------|------|
| Row1 | a    | b    |
| Row2 | c    | d    |

# Vectorized String Operations

- ▶ Pandas provides a comprehensive set of *vectorized string operations* via the **str** attribute of Pandas Series and Index object containing strings
- ▶ These vectorized string operations become most useful in the process of cleaning up messy, real-world data
- ▶ The operations correctly handle missing data (None values)
- ▶ For example, suppose we create a Pandas Series with the following data:

```
names = pd.Series(['peter', 'Paul', None, 'MARY', 'GUIDO'])
names
```

```
0 peter
1 Paul
2 None
3 MARY
4 GUIDO
dtype: object
```

# Vectorized String Operations

- ▶ We can now call a single method that will capitalize all the entries, while skipping over any missing values:

```
names.str.capitalize()
```

```
0 Peter
1 Paul
2 None
3 Mary
4 Guido
dtype: object
```

- ▶ Using tab completion on this str attribute will list all the vectorized string methods available to Pandas

# Methods Similar to Python String Methods

- ▶ Nearly all Python's built-in string methods are mirrored by a Pandas vectorized string method
- ▶ Here is a list of Pandas str methods that mirror Python string methods:

|          |              |              |              |
|----------|--------------|--------------|--------------|
| len()    | lower()      | translate()  | islower()    |
| ljust()  | upper()      | startswith() | isupper()    |
| rjust()  | find()       | endswith()   | isnumeric()  |
| center() | rfind()      | isalnum()    | isdecimal()  |
| zfill()  | index()      | isalpha()    | split()      |
| strip()  | rindex()     | isdigit()    | rsplit()     |
| rstrip() | capitalize() | isspace()    | partition()  |
| lstrip() | swapcase()   | istitle()    | rpartition() |

# Methods Similar to Python String Methods

- ▶ These methods have various return values
- ▶ Some, like lower(), return a series of strings:

```
monte = pd.Series(['Graham Chapman', 'John Cleese', 'Terry Gilliam',
 'Eric Idle', 'Terry Jones', 'Michael Palin'])
```

```
monte.str.lower()
```

```
0 graham chapman
1 john cleese
2 terry gilliam
3 eric idle
4 terry jones
5 michael palin
dtype: object
```

# Methods Similar to Python String Methods

- ▶ But some others return numbers:

```
monte.str.len()
```

```
0 14
1 11
2 13
3 9
4 11
5 13
dtype: int64
```

- ▶ Or Boolean values:

```
monte.str.startswith('T')
```

```
0 False
1 False
2 True
3 False
4 True
5 False
dtype: bool
```

# Methods Similar to Python String Methods

- ▶ Still others return lists or other compound values for each element:

```
monte.str.split()

0 [Graham, Chapman]
1 [John, Cleese]
2 [Terry, Gilliam]
3 [Eric, Idle]
4 [Terry, Jones]
5 [Michael, Palin]
dtype: object
```

- ▶ Passing **expand=True** to split() expands the splitted strings into separate columns:

```
monte.str.split(expand=True)
```

|   | 0       | 1       |
|---|---------|---------|
| 0 | Graham  | Chapman |
| 1 | John    | Cleese  |
| 2 | Terry   | Gilliam |
| 3 | Eric    | Idle    |
| 4 | Terry   | Jones   |
| 5 | Michael | Palin   |

# Miscellaneous Methods

- ▶ Finally, there are some miscellaneous methods that enable other operations:

| Method          | Description                                                       |
|-----------------|-------------------------------------------------------------------|
| get()           | Index each element                                                |
| slice()         | Slice each element                                                |
| slice_replace() | Replace slice in each element with passed value                   |
| cat()           | Concatenate strings                                               |
| repeat()        | Repeat values                                                     |
| normalize()     | Return Unicode form of string                                     |
| pad()           | Add whitespace to left, right, or both sides of strings           |
| wrap()          | Split long strings into lines with length less than a given width |
| join()          | Join strings in each element of the Series with passed separator  |
| get_dummies()   | extract dummy variables as a dataframe                            |

# Vectorized Item Access and Slicing

- ▶ The `get()` and `slice()` operations enable vectorized element access from each array
- ▶ For example, we can get a slice of the first three characters of each array using `df.str.slice(0, 3)`
- ▶ Note that this behavior is also available through Python's normal indexing syntax, e.g., `df.str.slice(0, 3)` is equivalent to `df.str[0:3]`:

```
monte.str[0:3]
```

```
0 Gra
1 Joh
2 Ter
3 Eri
4 Ter
5 Mic
dtype: object
```

- ▶ Indexing via `df.str.get(i)` and `df.str[i]` is likewise similar

# Vectorized Item Access and Slicing

- ▶ These get() and slice() methods also let you access elements of arrays returned by **split()**
- ▶ For example, to extract the last name of each entry, we can combine split() and get():

```
monte.str.split().str[-1]
```

```
0 Chapman
1 Cleese
2 Gilliam
3 Idle
4 Jones
5 Palin
dtype: object
```

# Indicator Variables

- ▶ The method `str.get_dummies()` is useful when your data has a column containing some sort of coded indicator
- ▶ For example, we might have a dataset that contains information in the form of codes, such as A="born in America", B="born in the United Kingdom", C="likes cheese", D="likes spam":

```
full_monte = pd.DataFrame({'name': monte,
 'info': ['B|C|D', 'B|D', 'A|C',
 'B|D', 'B|C', 'B|C|D']})
full_monte
```

|   | info  | name           |
|---|-------|----------------|
| 0 | B C D | Graham Chapman |
| 1 | B D   | John Cleese    |
| 2 | A C   | Terry Gilliam  |
| 3 | B D   | Eric Idle      |
| 4 | B C   | Terry Jones    |
| 5 | B C D | Michael Palin  |

# Indicator Variables

- ▶ The `str.get_dummies()` method lets you quickly split-out these indicator variables into a DataFrame:

```
full_monte['info'].str.get_dummies('|')
```

|   | A | B | C | D |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 2 | 1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 0 | 1 |
| 4 | 0 | 1 | 1 | 0 |
| 5 | 0 | 1 | 1 | 1 |

## Exercise

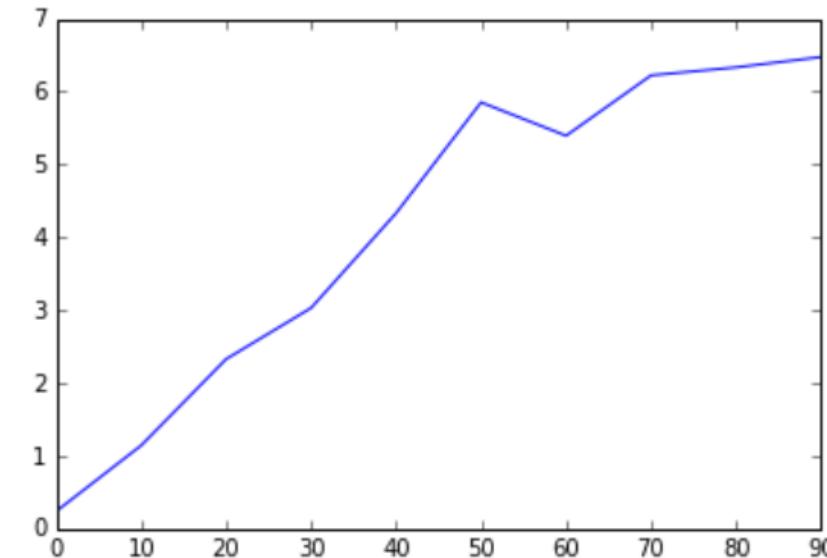
- ▶ Download the open recipes database from <https://github.com/sameergarg/scalasticsearch/blob/master/conf/recipeitems-latest.json.gz>
- ▶ The database is in JSON format
- ▶ Read the recipes data into a DataFrame
- ▶ Run the following queries on the data:
  - ▶ Find the name of the recipe that has the longest ingredient list
  - ▶ How many of the recipes are for breakfast food? (use the description column)
  - ▶ How many of the recipes contain precisely 3 ingredients?
- ▶ Write a function that given a list of ingredients, returns the names of all the recipes that use all those ingredients
- ▶ Test this function with the list ['parsley', 'paprika', 'tarragon']
  - ▶ You should get 10 recipes

# Plotting Functions in Pandas

- ▶ Pandas has a number of high-level plotting methods for creating standard visualizations that take advantage of how data is organized in DataFrame objects
- ▶ Series and DataFrame have a **plot()** method for making many different plot types
- ▶ By default, they make line plots:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('classic')
import pandas as pd
import numpy as np
```

```
ser = pd.Series(np.random.randn(10).cumsum(),
 index=np.arange(0, 100, 10))
ser.plot();
```



- ▶ The Series object's index is passed to matplotlib for plotting on the X axis

# Customizing Plot

- ▶ You can customize these plots by using one of the following arguments:
  - ▶ These arguments are passed through to the respective matplotlib plotting function

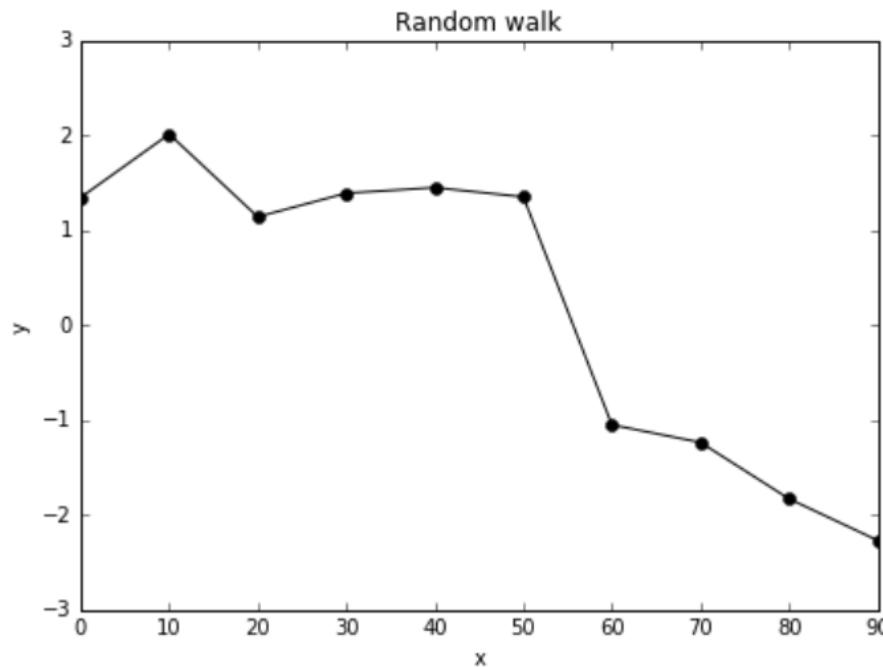
| Argument  | Description                                                                  |
|-----------|------------------------------------------------------------------------------|
| kind      | Can be 'line', 'bar', 'barh', 'hist', 'box', 'kde', 'density', 'area', 'pie' |
| ax        | matplotlib axes to plot on                                                   |
| figsize   | Size of figure to create as tuple                                            |
| use_index | Use the object index for tick labels                                         |
| title     | Title to use for the plot                                                    |
| grid      | Axis grid lines                                                              |
| legend    | Add a subplot legend (True by default)                                       |
| style     | Style string, like 'ko--', to be passed to matplotlib (one per column)       |
| logx      | Use log scaling on the X axis                                                |
| logy      | Use log scaling on the Y axis                                                |

| Argument | Description                                   |
|----------|-----------------------------------------------|
| xticks   | Values to use for X axis ticks                |
| yticks   | Values to use for Y axis ticks                |
| xlim,    | X axis limits (e.g. [0, 10])                  |
| ylim     | Y axis limits                                 |
| rot      | Rotation of tick labels (0 through 360)       |
| fontsize | Font size for xticks and yticks               |
| colormap | Colormap to select colors from                |
| xerr     | X axis error data                             |
| yerr     | Y axis error data                             |
| label    | label argument to provide to plot             |
| **kwds   | Options to pass to matplotlib plotting method |

# Customizing Plot

- In addition, the `plot()` function returns an `Axes` object, so you can further customize your plot using that object, e.g., to set the labels for the x and y axes:

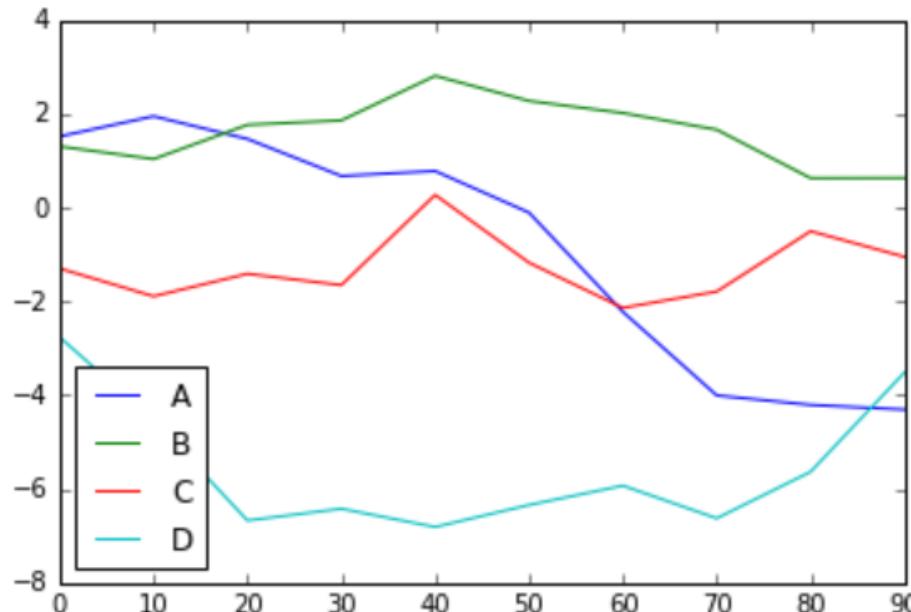
```
ser = pd.Series(np.random.randn(10).cumsum(),
 index=np.arange(0, 100, 10))
ax = ser.plot(title='Random walk', style=' -ok', figsize=(7, 5))
ax.set_xlabel('x')
ax.set_ylabel('y');
```



# Plotting a DataFrame

- ▶ DataFrame's plot() method plots each of its columns as a different line on the same subplot, creating a legend automatically

```
df = pd.DataFrame(np.random.randn(10, 4).cumsum(axis=0),
 columns=['A', 'B', 'C', 'D'],
 index=np.arange(0, 100, 10))
df.plot();
```



# Plotting a DataFrame

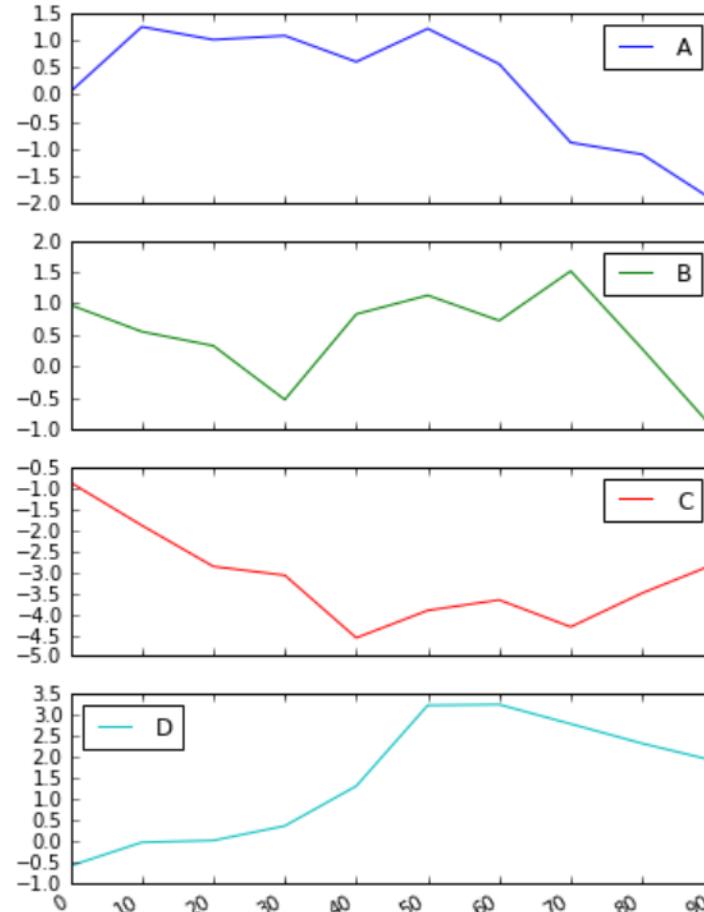
- ▶ DataFrame has a number of options allowing some flexibility with how the columns are handled
  - ▶ For example, whether to plot them all on the same subplot or to create separate subplots
- ▶ DataFrame-specific plot arguments:

| Argument     | Description                                                                                           |
|--------------|-------------------------------------------------------------------------------------------------------|
| subplots     | Plot each DataFrame column in a separate subplot                                                      |
| sharex       | If subplots=True, share the same X axis, linking ticks and limits                                     |
| sharey       | If subplots=True, share the same Y axis                                                               |
| sort_columns | Plot columns in alphabetical order; by default uses existing column order                             |
| secondary_y  | Whether to plot on the secondary y-axis<br>If a list/tuple, which columns to plot on secondary y-axis |

# Plotting a DataFrame

- ▶ Example for creating a separate subplot for each column:

```
df = pd.DataFrame(np.random.randn(10, 4).cumsum(axis=0),
 columns=['A', 'B', 'C', 'D'],
 index=np.arange(0, 100, 10))
df.plot(subplots=True, figsize=(6, 9));
```



# Plotting a DataFrame

- ▶ The DataFrame supports the following plot types via the kind argument:

| kind         | Description                    |
|--------------|--------------------------------|
| line         | line plot (default)            |
| bar          | vertical bar plot              |
| barh         | horizontal bar plot            |
| hist         | histogram                      |
| box          | boxplot                        |
| kde, density | Kernel Density Estimation plot |
| area         | area plot                      |
| pie          | pie plot                       |
| scatter      | scatter plot                   |

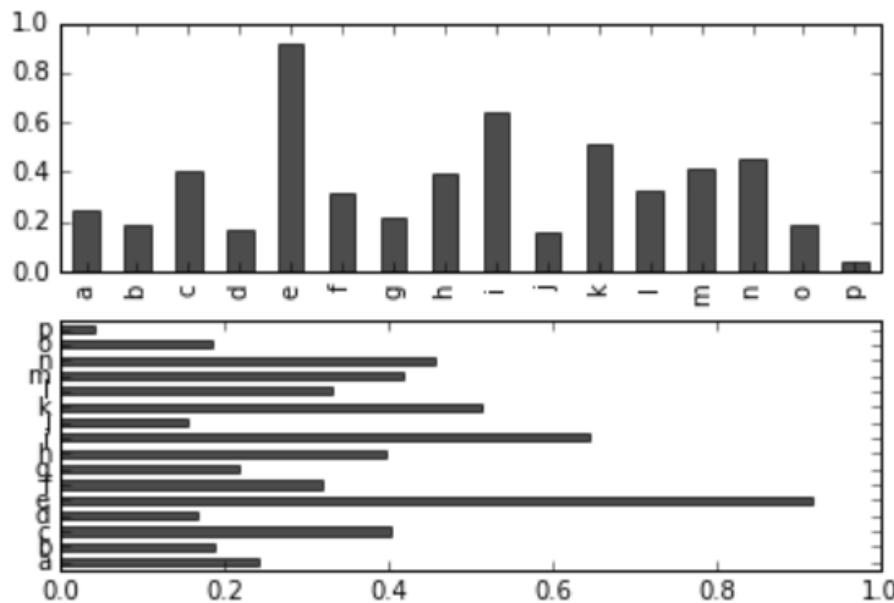
- ▶ Each plot kind has a corresponding method on the DataFrame.plot accessor, e.g., df.plot(kind='scatter') is equivalent to df.plot.scatter()

# Bar Plots

- In bar plots, the Series or DataFrame index will be used as the X (bar) or Y (barh) ticks

```
fig, axes = plt.subplots(2, 1)
data = pd.Series(np.random.rand(16), index=list('abcdefghijklmnope'))

data.plot.bar(ax=axes[0], color='k', alpha=0.7)
data.plot.barh(ax=axes[1], color='k', alpha=0.7);
```



# Bar Plots

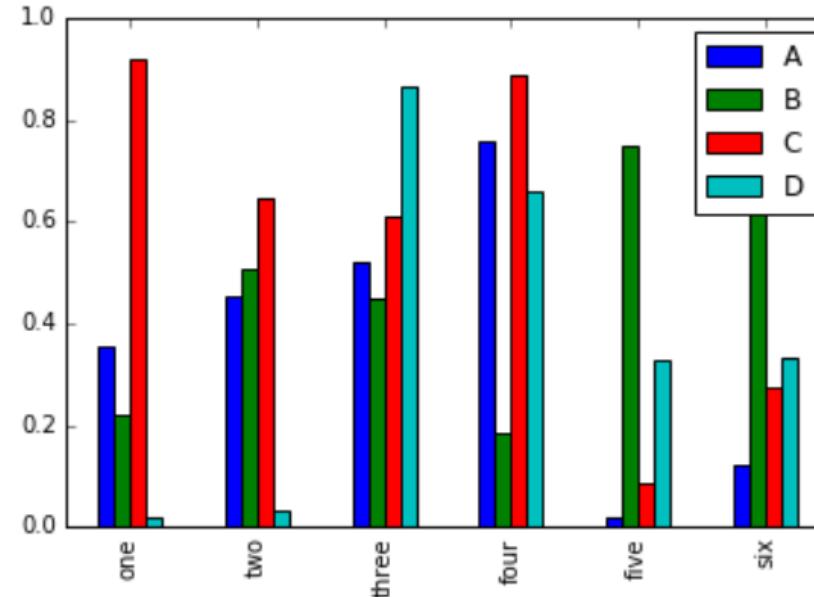
- With a DataFrame, bar plots group the values in each row together in a group in bars, side by side, for each value:

```
df = pd.DataFrame(np.random.rand(6, 4),
 index=['one', 'two', 'three', 'four',
 'five', 'six'],
 columns=['A', 'B', 'C', 'D'])

df
```

|       | A        | B        | C        | D        |
|-------|----------|----------|----------|----------|
| one   | 0.731235 | 0.882648 | 0.890212 | 0.711207 |
| two   | 0.593067 | 0.290674 | 0.449996 | 0.218162 |
| three | 0.468419 | 0.325028 | 0.434517 | 0.162489 |
| four  | 0.803957 | 0.652598 | 0.311046 | 0.281698 |
| five  | 0.738825 | 0.155606 | 0.976214 | 0.032735 |
| six   | 0.185331 | 0.288600 | 0.205113 | 0.176076 |

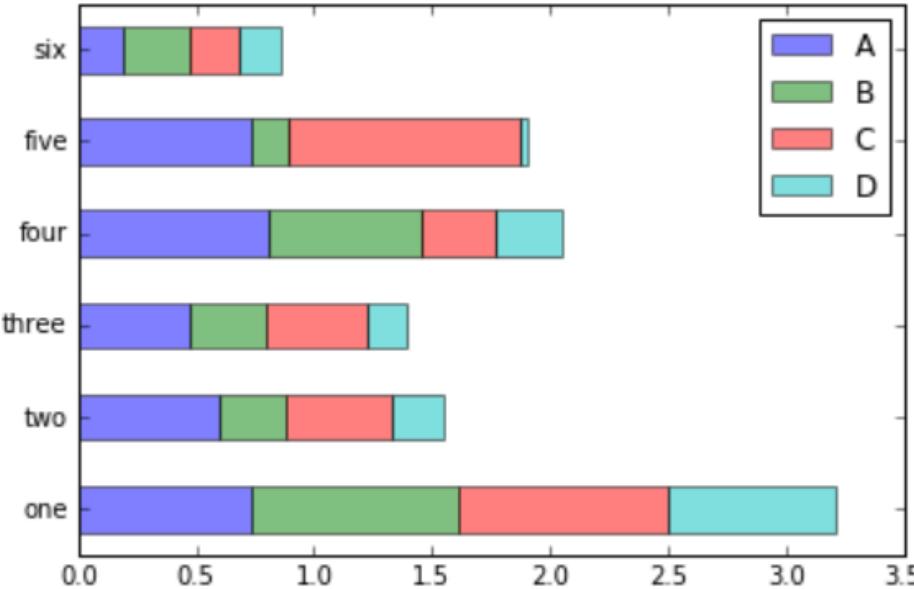
```
df.plot.bar();
```



# Bar Plots

- ▶ Stacked bar plots are created from a DataFrame by passing stacked=True, resulting in the value in each row being stacked together:

```
df.plot.barh(stacked=True, alpha=0.5);
```

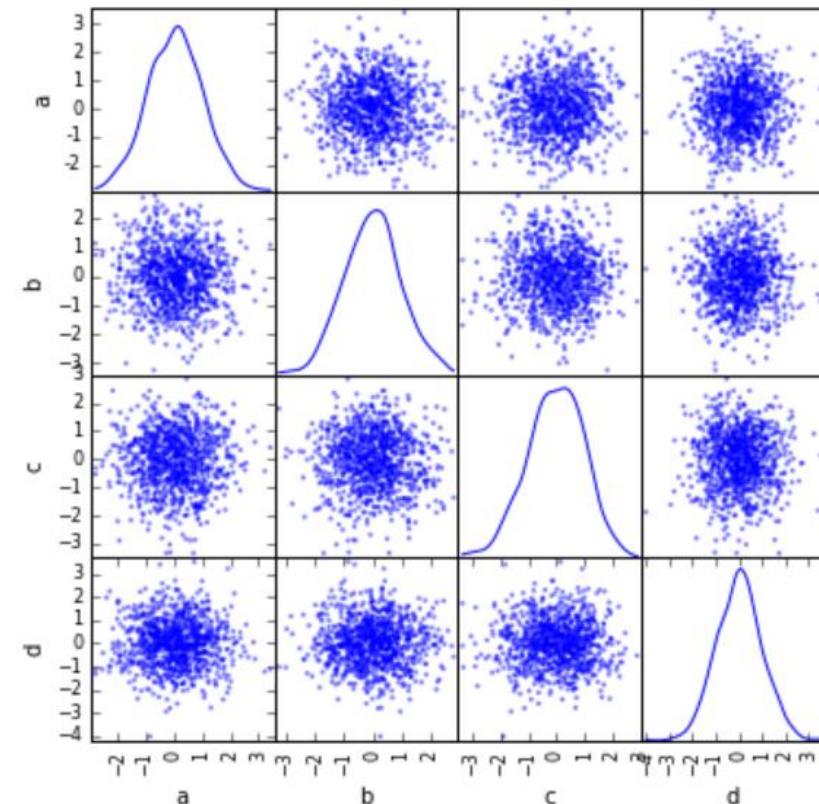


# Scatter Matrix Plot

- You can create a scatter plot matrix using the **scatter\_matrix()** method in the module **pandas.plotting**:

```
from pandas.plotting import scatter_matrix

df = pd.DataFrame(np.random.randn(1000, 4),
 columns=['a', 'b', 'c', 'd'])
scatter_matrix(df, alpha=0.5, figsize=(6, 6), diagonal='kde');
```



# Combining and Merging Datasets

- ▶ Some of the most interesting studies of data come from combining different data sources
- ▶ Data contained in pandas objects can be combined together in a number of built-in ways:
  - ▶ **pandas.concat()** glues or stacks together objects along an axis
  - ▶ **pandas.merge()** connects rows in DataFrames based on one or more keys
    - ▶ This will be familiar to users of SQL or other relational databases, as it implements database *join operations*

## Recall: Concatenation of NumPy Arrays

- ▶ Concatenation of Series and DataFrame objects is very similar to concatenation of Numpy arrays, which can be done via the **np.concatenate()** function:

```
x = [1, 2, 3]
y = [4, 5, 6]
z = [7, 8, 9]
np.concatenate([x, y, z])
```

```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

- ▶ The first argument is a list or tuple of arrays to concatenate
- ▶ Additionally, it takes an axis keyword that allows you to specify the axis along which the result will be concatenated:

```
x = [[1, 2],
 [3, 4]]
np.concatenate([x, x], axis=1)
```

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

# Simple Concatenation

- ▶ **pd.concat()** has a similar syntax to np.concatenate(), but contains a number of additional options:

```
pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False,
 keys=None, levels=None, names=None, verify_integrity=False,
 copy=True)
```

- ▶ For example, it can be used for a simple concatenation of two Series:

```
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([ser1, ser2])
```

```
1 A
2 B
3 C
4 D
5 E
6 F
dtype: object
```

# Simple Concatenation

- ▶ It also works to concatenate higher-dimensional objects, such as DataFrames:

```
df1 = make_df('AB', [1, 2])
df2 = make_df('AB', [3, 4])
concat_df1_df2 = pd.concat([df1, df2])
display('df1', 'df2', 'concat_df1_df2')
```

df1                  df2                  concat\_df1\_df2

| A    B |    | A    B |   | A    B |    |
|--------|----|--------|---|--------|----|
| 1      | A1 | B1     | 3 | A3     | B3 |
| 2      | A2 | B2     | 4 | A4     | B4 |
|        |    |        | 2 | A2     | B2 |
|        |    |        | 3 | A3     | B3 |
|        |    |        | 4 | A4     | B4 |

# Simple Concatenation

- ▶ By default, the concatenation takes place row-wise within the DataFrame (axis=0)
- ▶ Like np.concatenate, pd.concat() allows specification of an **axis** along which concatenation will take place:

```
df3 = make_df('AB', [0, 1])
df4 = make_df('CD', [0, 1])
concat_df3_df4 = pd.concat([df3, df4], axis=1)
display('df3', 'df4', 'concat_df3_df4')
```

|   | df3 |    | df4 |    | concat_df3_df4 |   |    |    |    |    |
|---|-----|----|-----|----|----------------|---|----|----|----|----|
|   | A   | B  | C   | D  | A              | B | C  | D  |    |    |
| 0 | A0  | B0 | 0   | C0 | D0             | 0 | A0 | B0 | C0 | D0 |
| 1 | A1  | B1 | 1   | C1 | D1             | 1 | A1 | B1 | C1 | D1 |

# Concatenation with Joins

- In practice, data from different sources might have different sets of column names, and pd.concat() offers several options in this case
- Consider the concatenation of the following two DataFrames, which have some (but not all!) columns in common:

```
df5 = make_df('ABC', [1, 2])
df6 = make_df('BCD', [3, 4])
concat_df5_df6 = pd.concat([df5, df6])
display('df5', 'df6', 'concat_df5_df6')
```

df5

|   | A  | B  | C  |
|---|----|----|----|
| 1 | A1 | B1 | C1 |
| 2 | A2 | B2 | C2 |

df6

|   | B  | C  | D  |
|---|----|----|----|
| 3 | B3 | C3 | D3 |
| 4 | B4 | C4 | D4 |

concat\_df5\_df6

|   | A   | B  | C  | D   |
|---|-----|----|----|-----|
| 1 | A1  | B1 | C1 | NaN |
| 2 | A2  | B2 | C2 | NaN |
| 3 | NaN | B3 | C3 | D3  |
| 4 | NaN | B4 | C4 | D4  |

# Concatenation with Joins

- ▶ By default, the entries for which no data is available are filled with NA values
- ▶ To change this, we can use the **join** and **join\_axes** parameters of pd.concat()
- ▶ By default, the join is a union of the input columns (join='outer'), but we can change this to an intersection of the columns using join='inner':

```
concat_df5_df6 = pd.concat([df5, df6], join='inner')
display('df5', 'df6', 'concat_df5_df6')
```

| df5 |    |    | df6 |   |    |    | concat_df5_df6 |   |       |
|-----|----|----|-----|---|----|----|----------------|---|-------|
|     | A  | B  | C   | B | C  | D  | B              | C |       |
| 1   | A1 | B1 | C1  | 3 | B3 | C3 | D3             | 1 | B1 C1 |
| 2   | A2 | B2 | C2  | 4 | B4 | C4 | D4             | 2 | B2 C2 |
|     |    |    |     |   |    |    |                | 3 | B3 C3 |
|     |    |    |     |   |    |    |                | 4 | B4 C4 |

# Exercise

- ▶ Predict the result of the following operations:

```
s1 = pd.Series([0, 1], index=['a', 'b'])
s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e'])
s3 = pd.Series([5, 6], index=['f', 'g'])
```

```
pd.concat([s1, s2, s3])
```

```
pd.concat([s1, s2, s3], axis=1)
```

```
s4 = pd.concat([s1 * 5, s3])
pd.concat([s1, s4], axis=1)
```

```
pd.concat([s1, s4], axis=1, join='inner')
```

```
pd.concat([s1, s4], axis=1, join_axes=[['a', 'c', 'b', 'e']])
```

```
pd.concat([s1, s1, s3], keys=['one', 'two', 'three'])
```

```
pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'three'])
```

# Merging Datasets

- ▶ One essential feature offered by Pandas is its high-performance, in-memory join and merge operations
- ▶ Merge or join operations combine datasets by linking rows using one or more keys
- ▶ These operations are central to relational databases
- ▶ The **pd.merge()** function is the main entry point for using these algorithms
- ▶ **pd.merge()** implements a number of types of joins:
  - ▶ one-to-one
  - ▶ many-to-one
  - ▶ many-to-many

# One-To-One Joins

- ▶ The simplest type of merge situation is the **one-to-one** join, in which each DataFrame has one value for each key
- ▶ Consider the following two DataFrames which contain information on several employees in a company:

```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
 'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
 'hire_date': [2004, 2008, 2012, 2014]})
display('df1', 'df2')
```

df1

|   | employee | group       |
|---|----------|-------------|
| 0 | Bob      | Accounting  |
| 1 | Jake     | Engineering |
| 2 | Lisa     | Engineering |
| 3 | Sue      | HR          |

df2

|   | employee | hire_date |
|---|----------|-----------|
| 0 | Lisa     | 2004      |
| 1 | Bob      | 2008      |
| 2 | Jake     | 2012      |
| 3 | Sue      | 2014      |

# One-To-One Joins

- ▶ We can use `pd.merge()` to combine this information into a single DataFrame:

```
df3 = pd.merge(df1, df2)
df3
```

|   | employee | group       | hire_date |
|---|----------|-------------|-----------|
| 0 | Bob      | Accounting  | 2008      |
| 1 | Jake     | Engineering | 2012      |
| 2 | Lisa     | Engineering | 2004      |
| 3 | Sue      | HR          | 2014      |

- ▶ `pd.merge()` recognizes that each DataFrame has an "employee" column, and automatically joins using this column as a key
  - ▶ If not specified, `pd.merge()` uses the overlapping column names as the keys
- ▶ The order of entries in each column is not necessarily maintained, and generally the merge discards the index (unless performing merges by index)

# Many-To-One Joins

- ▶ Joins in which one of the two key columns contains duplicates
- ▶ The resulting DataFrame will preserve those duplicate entries as appropriate
- ▶ For example:

```
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
 'supervisor': ['Carly', 'Guido', 'Steve']})
display('df3', 'df4', 'pd.merge(df3, df4)')
```

| df3 |          |             | df4       |   | pd.merge(df3, df4) |            |          |       |             |            |
|-----|----------|-------------|-----------|---|--------------------|------------|----------|-------|-------------|------------|
|     | employee | group       | hire_date |   | group              | supervisor | employee | group | hire_date   | supervisor |
| 0   | Bob      | Accounting  | 2008      | 0 | Accounting         | Carly      | 0        | Bob   | Accounting  | 2008       |
| 1   | Jake     | Engineering | 2012      | 1 | Engineering        | Guido      | 1        | Jake  | Engineering | Guido      |
| 2   | Lisa     | Engineering | 2004      | 2 | HR                 | Steve      | 2        | Lisa  | Engineering | Guido      |
| 3   | Sue      | HR          | 2014      |   |                    |            | 3        | Sue   | HR          | Steve      |

# Many-To-Many Joins

- ▶ Joins in which both the left and right DataFrames contain duplicates
- ▶ Consider the following, where we have a DataFrame showing one or more skills associated with a particular group:

```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
 'group': ['Accounting', 'Engineering', 'Engineering', 'HR'],
 'skills': ['math', 'spreadsheets', 'coding', 'linux']})
display('df1', 'df5', "pd.merge(df1, df5)")

df5 = pd.DataFrame({'group': ['Accounting', 'Accounting', 'Engineering', 'Engineering', 'HR'],
 'skills': ['math', 'spreadsheets', 'coding', 'linux', 'spreadsheets', 'organization']})
display('df1', 'df5', "pd.merge(df1, df5)")
```

| df1      |       | df5         |        | pd.merge(df1, df5) |              |        |      |             |              |
|----------|-------|-------------|--------|--------------------|--------------|--------|------|-------------|--------------|
| employee | group | group       | skills | employee           | group        | skills |      |             |              |
| 0        | Bob   | Accounting  | 0      | Accounting         | math         | 0      | Bob  | Accounting  | math         |
| 1        | Jake  | Engineering | 1      | Accounting         | spreadsheets | 1      | Bob  | Accounting  | spreadsheets |
| 2        | Lisa  | Engineering | 2      | Engineering        | coding       | 2      | Jake | Engineering | coding       |
| 3        | Sue   | HR          | 3      | Engineering        | linux        | 3      | Jake | Engineering | linux        |
|          |       |             | 4      | HR                 | spreadsheets | 4      | Lisa | Engineering | coding       |
|          |       |             | 5      | HR                 | organization | 5      | Lisa | Engineering | linux        |
|          |       |             |        |                    |              | 6      | Sue  | HR          | spreadsheets |
|          |       |             |        |                    |              | 7      | Sue  | HR          | organization |

# Specifying the Merge Key

- ▶ By default `pd.merge()` looks for one or more matching column names between the two inputs, and uses this as the key
- ▶ However, often the column names will not match so nicely, and `pd.merge()` provides a variety of options for handling this
- ▶ Most simply, you can explicitly specify the name of the key column using the **on** keyword, which takes a column name or a list of column names:

```
df3 = pd.merge(df1, df2, on='employee')
display('df1', 'df2', 'df3')
```

| df1 |          | df2         |   | df3      |           |          |       |             |
|-----|----------|-------------|---|----------|-----------|----------|-------|-------------|
|     | employee | group       |   | employee | hire_date | employee | group | hire_date   |
| 0   | Bob      | Accounting  | 0 | Lisa     | 2004      | 0        | Bob   | Accounting  |
| 1   | Jake     | Engineering | 1 | Bob      | 2008      | 1        | Jake  | Engineering |
| 2   | Lisa     | Engineering | 2 | Jake     | 2012      | 2        | Lisa  | Engineering |
| 3   | Sue      | HR          | 3 | Sue      | 2014      | 3        | Sue   | HR          |

# Specifying the Merge Key

- ▶ At times you may wish to merge two datasets with different column names
- ▶ For example, we may have a dataset in which the employee name is labeled as "name" rather than "employee"
- ▶ In this case, we can use the **left\_on** and **right\_on** keywords to specify the two column names:

```
df6 = pd.DataFrame({'name': ['Jake', 'Sue', 'Lisa', 'Bob'],
 'salary': [70000, 80000, 120000, 90000]})
df7 = pd.merge(df1, df6, left_on='employee', right_on='name')
display('df1', 'df6', 'df7')
```

| df1 |          |             | df6 |      | df7    |   |          |             |
|-----|----------|-------------|-----|------|--------|---|----------|-------------|
|     | employee | group       |     | name | salary |   | employee | group       |
| 0   | Bob      | Accounting  | 0   | Jake | 70000  | 0 | Bob      | Accounting  |
| 1   | Jake     | Engineering | 1   | Sue  | 80000  | 1 | Jake     | Engineering |
| 2   | Lisa     | Engineering | 2   | Lisa | 120000 | 2 | Lisa     | Engineering |
| 3   | Sue      | HR          | 3   | Bob  | 90000  | 3 | Sue      | HR          |

# Overlapping Column Names

- Finally, you may end up in a case where your two input DataFrames have conflicting column names

```
df10 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
 'rank': [1, 2, 3, 4]})
df11 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
 'rank': [3, 1, 4, 2]})
display('df10', 'df11', 'pd.merge(df10, df11, on="name")')
```

|   | df10      | df11      | pd.merge(df10, df11, on="name") |
|---|-----------|-----------|---------------------------------|
|   | name rank | name rank | name rank_x rank_y              |
| 0 | Bob 1     | 0 Bob 3   | 0 Bob 1 3                       |
| 1 | Jake 2    | 1 Jake 1  | 1 Jake 2 1                      |
| 2 | Lisa 3    | 2 Lisa 4  | 2 Lisa 3 4                      |
| 3 | Sue 4     | 3 Sue 2   | 3 Sue 4 2                       |

- Because the output would have two conflicting column names, the merge function automatically appends a suffix \_x or \_y to make the output columns unique

# Overlapping Column Names

- If these defaults are inappropriate, it is possible to specify a custom suffix using the **suffixes** keyword:

```
pd.merge(df10, df11, on='name', suffixes=['_L', '_R'])
```

|   | name | rank_L | rank_R |
|---|------|--------|--------|
| 0 | Bob  | 1      | 3      |
| 1 | Jake | 2      | 1      |
| 2 | Lisa | 3      | 4      |
| 3 | Sue  | 4      | 2      |

- These suffixes work in any of the possible join patterns, and work also if there are multiple overlapping columns

## Exercise

- ▶ Download the following data files containing some data about US states and their populations from <https://goo.gl/KnFPfC>
  - ▶ state-population.csv
  - ▶ state-areas.csv
  - ▶ state-abbrevs.csv
- ▶ Use Pandas to rank the states by their 2010 total population density from the most dense state to the least dense state
- ▶ The first 5 rows of the result should look like this:

|   | state                | population | area (sq. mi) | density     |
|---|----------------------|------------|---------------|-------------|
| 0 | District of Columbia | 605125.0   | 68            | 8898.897059 |
| 1 | Puerto Rico          | 3721208.0  | 3515          | 1058.665149 |
| 2 | New Jersey           | 8802707.0  | 8722          | 1009.253268 |
| 3 | Rhode Island         | 1052669.0  | 1545          | 681.339159  |
| 4 | Connecticut          | 3579210.0  | 5544          | 645.600649  |

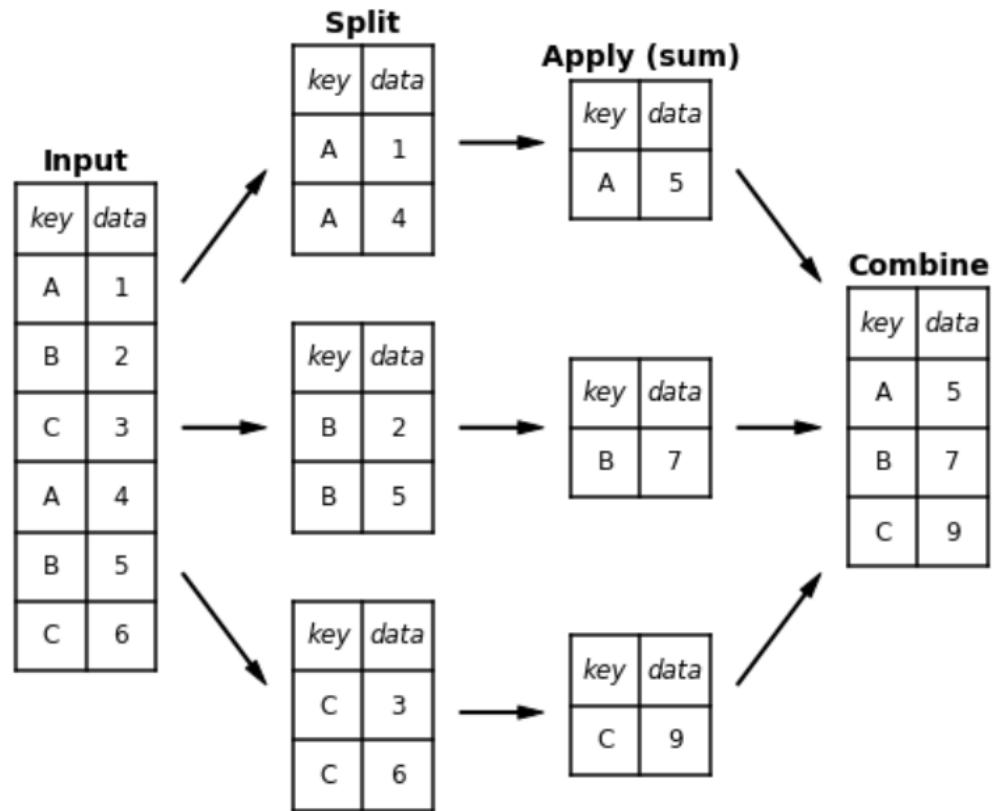
# Aggregation and Grouping

---

- ▶ Categorizing a data set and applying a function to each group, whether an aggregation or transformation, is often a critical component of a data analysis workflow
- ▶ After loading, merging, and preparing a data set, a familiar task is to compute group statistics or possibly pivot tables for reporting or visualization purposes
- ▶ Pandas provides a flexible and high-performance groupby facility, enabling you to slice and dice, and summarize data sets in a natural way

# GroupBy

- Group operations can be described by the **split-apply-combine** process:



The *split* step breaks up and group a DataFrame depending on the value of the specified key.

The *apply* step involves computing some function, usually an aggregate, transformation, or filtering, within the individual groups.

The *combine* step merges the results of these operations into an output array.

# GroupBy

- ▶ GroupBy typically performs all these steps in a single pass over the data
- ▶ The user need not think about *how* the computation is done under the hood, but rather think about the *operation as a whole*
- ▶ As a concrete example, let's take a look at using Pandas for the computation shown in this diagram. We'll start by creating the input DataFrame:

```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
 'data': range(1, 7)},
 columns=['key', 'data'])
```

df

|   | key | data |
|---|-----|------|
| 0 | A   | 1    |
| 1 | B   | 2    |
| 2 | C   | 3    |
| 3 | A   | 4    |
| 4 | B   | 5    |
| 5 | C   | 6    |

# GroupBy

- ▶ The most basic group by operation can be computed with the **groupby()** method of DataFrame, passing the name of the desired key column:

```
df.groupby('key')
```

```
<pandas.core.groupby.DataFrameGroupBy object at 0x000001C2E7A0ECF8>
```

- ▶ The result is a DataFrameGroupBy object, which is a special view of the DataFrame, that has all of the information needed to then apply some operation to each group
- ▶ To produce a result, we can apply an aggregate to this object, which will perform the appropriate apply/combine steps to produce the desired result:

```
df.groupby('key').sum()
```

|     | data |
|-----|------|
| key |      |
| A   | 5    |
| B   | 7    |
| C   | 9    |

# The GroupBy Object

- ▶ The GroupBy object is a very flexible abstraction
- ▶ In many ways, you can treat it as if it's a collection of DataFrames
- ▶ The most important operations made available by a GroupBy are *aggregate*, *filter*, *transform*, and *apply*
- ▶ Let's see some examples using the Planets data, available via the seaborn package

# Planets Data

- ▶ The Planets dataset gives information on planets that astronomers have discovered around other stars
- ▶ It can be downloaded with a simple Seaborn command:

```
import seaborn as sns
planets = sns.load_dataset('planets')
planets.shape
```

```
(1035, 6)
```

```
planets.head()
```

|   | method          | number | orbital_period | mass  | distance | year |
|---|-----------------|--------|----------------|-------|----------|------|
| 0 | Radial Velocity | 1      | 269.300        | 7.10  | 77.40    | 2006 |
| 1 | Radial Velocity | 1      | 874.774        | 2.21  | 56.95    | 2008 |
| 2 | Radial Velocity | 1      | 763.000        | 2.60  | 19.84    | 2011 |
| 3 | Radial Velocity | 1      | 326.030        | 19.40 | 110.62   | 2007 |
| 4 | Radial Velocity | 1      | 516.220        | 10.50 | 119.47   | 2009 |

- ▶ This has some details on the 1,000+ extrasolar planets discovered up to 2014

# Column Indexing

- ▶ The GroupBy object supports column indexing in the same way as the DataFrame, and returns a modified GroupBy object:

```
planets.groupby('method')
```

```
<pandas.core.groupby.DataFrameGroupBy object at 0x000001C2E9B95860>
```

```
planets.groupby('method')['orbital_period']
```

```
<pandas.core.groupby.SeriesGroupBy object at 0x000001C2E9B810B8>
```

- ▶ Here we've selected a particular Series group from the original DataFrame group by reference to its column name

# Column Indexing

- As with the GroupBy object, no computation is done until we call some aggregate on the object:

```
planets.groupby('method')['orbital_period'].median()
```

```
method
Astrometry 631.180000
Eclipse Timing Variations 4343.500000
Imaging 27500.000000
Microlensing 3300.000000
Orbital Brightness Modulation 0.342887
Pulsar Timing 66.541900
Pulsation Timing Variations 1170.000000
Radial Velocity 360.200000
Transit 5.714932
Transit Timing Variations 57.011000
Name: orbital_period, dtype: float64
```

- This gives an idea of the general scale of orbital periods (in days) that each method of discovery is sensitive to

# Aggregation

- ▶ We're already familiar with GroupBy aggregations such as sum(), median(), etc.
- ▶ But the aggregate() method allows for even more flexibility
- ▶ It can take a string, a function, or a list thereof, and compute all the aggregates at once
- ▶ Here is a quick example combining all these:

```
df.groupby('key').aggregate(['min', np.median, max])
```

|     | data1 |        |     | data2 |        |     |
|-----|-------|--------|-----|-------|--------|-----|
|     | min   | median | max | min   | median | max |
| key |       |        |     |       |        |     |
| A   | 0     | 1.5    | 3   | 3     | 4.0    | 5   |
| B   | 1     | 2.5    | 4   | 0     | 3.5    | 7   |
| C   | 2     | 3.5    | 5   | 3     | 6.0    | 9   |

# Aggregation

- ▶ Another useful pattern is to pass a dictionary mapping column names to operations to be applied on that column:

```
df.groupby('key').aggregate({'data1': 'min',
 'data2': 'max'})
```

|     | data1 | data2 |
|-----|-------|-------|
| key |       |       |
| A   | 0     | 5     |
| B   | 1     | 7     |
| C   | 2     | 9     |

# Filtering

- ▶ A filtering operation allows you to drop data based on the group properties
- ▶ For example, we might want to keep all groups in which the standard deviation is larger than some critical value:

```
def filter_func(x):
 return x['data2'].std() > 4

filtered = df.groupby('key').filter(filter_func)
display('df', "df.groupby('key').std()", 'filtered')
```

| df |     |       | df.groupby('key').std() |  | filtered |         |          |
|----|-----|-------|-------------------------|--|----------|---------|----------|
|    | key | data1 | data2                   |  | key      | data1   | data2    |
| 0  | A   | 0     | 5                       |  | A        | 2.12132 | 1.414214 |
| 1  | B   | 1     | 0                       |  | B        | 2.12132 | 4.949747 |
| 2  | C   | 2     | 3                       |  | C        | 2.12132 | 4.242641 |
| 3  | A   | 3     | 3                       |  |          |         |          |
| 4  | B   | 4     | 7                       |  |          |         |          |
| 5  | C   | 5     | 9                       |  |          |         |          |

# Transformation

- ▶ While aggregation must return a reduced version of the data, transformation can return some transformed version of the full data to recombine
- ▶ The method **transform()** gets a Series containing the group elements and returns the same Series after transformation
- ▶ A common example is to center the data by subtracting the group-wise mean:

```
df.groupby('key').transform(lambda x: x - x.mean())
```

|   | data1 | data2 |
|---|-------|-------|
| 0 | -1.5  | 1.0   |
| 1 | -1.5  | -3.5  |
| 2 | -1.5  | -3.0  |
| 3 | 1.5   | -1.0  |
| 4 | 1.5   | 3.5   |
| 5 | 1.5   | 3.0   |

# The apply() Method

- ▶ The apply() method lets you apply an arbitrary function to the group results
- ▶ The function should take a DataFrame, and return either a Pandas object (e.g., DataFrame, Series) or a scalar
- ▶ The combine operation will be tailored to the type of output returned.
- ▶ For example, we can normalize the first column by the sum of the second:

```
def norm_by_data2(x):
 # x is a DataFrame of group values
 x['data1'] /= x['data2'].sum()
 return x

normalized = df.groupby('key').apply(norm_by_data2)
normalized
display('df', 'normalized')
```

| df |     |       |       | normalized |          |       |
|----|-----|-------|-------|------------|----------|-------|
|    | key | data1 | data2 | key        | data1    | data2 |
| 0  | A   | 0     | 5     | 0          | 0.000000 | 5     |
| 1  | B   | 1     | 0     | 1          | 0.142857 | 0     |
| 2  | C   | 2     | 3     | 2          | 0.166667 | 3     |
| 3  | A   | 3     | 3     | 3          | 0.375000 | 3     |
| 4  | B   | 4     | 7     | 4          | 0.571429 | 7     |
| 5  | C   | 5     | 9     | 5          | 0.416667 | 9     |

# Specifying the Split Key

- ▶ So far we split the DataFrame on a single column name
- ▶ This is just one of many options by which the groups can be defined
- ▶ For example, you can specify a list of column names to group by:

```
planets.groupby(['method', 'year'])['distance'].mean()
```

| method                    | year | distance   |
|---------------------------|------|------------|
| Astrometry                | 2010 | 14.980000  |
|                           | 2013 | 20.770000  |
| Eclipse Timing Variations | 2008 | 130.720000 |
|                           | 2009 | NaN        |
|                           | 2010 | 500.000000 |
|                           | 2011 | NaN        |
|                           | 2012 | NaN        |
| Imaging                   | 2004 | 110.466667 |
|                           | 2005 | 45.520000  |
|                           | 2006 | 29.620000  |
|                           | 2007 | 165.000000 |

# Specifying the Split Key

- If you want to be more specific, the split key can be any series or list with a length matching that of the DataFrame:

```
L = [0, 1, 0, 1, 2, 0]
display('df', 'df.groupby(L).sum()')
```

df                    df.groupby(L).sum()

|   | key | data1 |   | data2 |    |
|---|-----|-------|---|-------|----|
|   |     | 0     | 1 | 0     | 17 |
| 0 | A   | 0     | 5 |       |    |
| 1 | B   | 1     | 0 | 1     | 3  |
| 2 | C   | 2     | 3 | 2     | 7  |
| 3 | A   | 3     | 3 |       |    |
| 4 | B   | 4     | 7 |       |    |
| 5 | C   | 5     | 9 |       |    |

# Specifying the Split Key

- ▶ Another method is to provide a dictionary that maps **index** values to the group keys:

```
df2 = df.set_index('key')
mapping = {'A': 'vowel', 'B': 'consonant', 'C': 'consonant'}
display('df2', 'df2.groupby(mapping).sum()')
```

df2                    df2.groupby(mapping).sum()

| key | data1     |       | data2 |    |
|-----|-----------|-------|-------|----|
|     | consonant | vowel | 12    | 19 |
| A   | 0         | 5     |       |    |
| B   | 1         | 0     |       |    |
| C   | 2         | 3     |       |    |
| A   | 3         | 3     |       |    |
| B   | 4         | 7     |       |    |
| C   | 5         | 9     |       |    |

# Specifying the Split Key

- ▶ Similar to mapping, you can pass any Python function that will input the **index** value and output the group:

```
display('df2', 'df2.groupby(str.lower).mean()')
```

```
df2 df2.groupby(str.lower).mean()
```

|     | data1 | data2 |   | data1 | data2 |
|-----|-------|-------|---|-------|-------|
| key |       |       |   | a     | 1.5   |
| A   | 0     | 5     | b | 2.5   | 3.5   |
| B   | 1     | 0     | c | 3.5   | 6.0   |
| C   | 2     | 3     |   |       |       |
| A   | 3     | 3     |   |       |       |
| B   | 4     | 7     |   |       |       |
| C   | 5     | 9     |   |       |       |

# Specifying the Split Key

- ▶ Further, any of the preceding key choices can be combined to group on a multi-index:

```
df2.groupby([str.lower, mapping]).mean()
```

|   |           | data1 | data2 |
|---|-----------|-------|-------|
| a | vowel     | 1.5   | 4.0   |
| b | consonant | 2.5   | 3.5   |
| c | consonant | 3.5   | 6.0   |

# Exercise

- ▶ Using the planet data, count the discovered planets by method and by decade
- ▶ Your output should look like this:

|                                      | decade | 1980s | 1990s | 2000s | 2010s |
|--------------------------------------|--------|-------|-------|-------|-------|
|                                      | method |       |       |       |       |
| <b>Astrometry</b>                    |        | 0.0   | 0.0   | 0.0   | 2.0   |
| <b>Eclipse Timing Variations</b>     |        | 0.0   | 0.0   | 5.0   | 10.0  |
| <b>Imaging</b>                       |        | 0.0   | 0.0   | 29.0  | 21.0  |
| <b>Microlensing</b>                  |        | 0.0   | 0.0   | 12.0  | 15.0  |
| <b>Orbital Brightness Modulation</b> |        | 0.0   | 0.0   | 0.0   | 5.0   |
| <b>Pulsar Timing</b>                 |        | 0.0   | 9.0   | 1.0   | 1.0   |
| <b>Pulsation Timing Variations</b>   |        | 0.0   | 0.0   | 1.0   | 0.0   |
| <b>Radial Velocity</b>               |        | 1.0   | 52.0  | 475.0 | 424.0 |
| <b>Transit</b>                       |        | 0.0   | 0.0   | 64.0  | 712.0 |
| <b>Transit Timing Variations</b>     |        | 0.0   | 0.0   | 0.0   | 9.0   |

# Pivot Tables

- ▶ We have seen how GroupBy lets us explore relationships within a dataset
- ▶ A *pivot table* is a similar operation that is commonly seen in spreadsheets and other programs that operate on tabular data
- ▶ The pivot table takes simple column-wise data as input, and groups the entries into a two-dimensional table that provides a multidimensional summarization of the data
- ▶ You can think of pivot tables as a multidimensional version of GroupBy aggregation
- ▶ That is, you split-apply-combine, but both the split and the combine happen across not a one-dimensional index, but across a two-dimensional grid

# Motivating Pivot Tables

- For the illustration of pivot tables, we'll use the database of passengers on the *Titanic*, available through the Seaborn library:

```
import numpy as np
import pandas as pd
import seaborn as sns
```

```
titanic = sns.load_dataset('titanic')
titanic.head()
```

|   | survived | pclass | sex    | age  | sibsp | parch | fare    | embarked | class | who   | adult_male | de |
|---|----------|--------|--------|------|-------|-------|---------|----------|-------|-------|------------|----|
| 0 | 0        | 3      | male   | 22.0 | 1     | 0     | 7.2500  | S        | Third | man   | True       | N  |
| 1 | 1        | 1      | female | 38.0 | 1     | 0     | 71.2833 | C        | First | woman | False      |    |
| 2 | 1        | 3      | female | 26.0 | 0     | 0     | 7.9250  | S        | Third | woman | False      | N  |
| 3 | 1        | 1      | female | 35.0 | 1     | 0     | 53.1000 | S        | First | woman | False      |    |
| 4 | 0        | 3      | male   | 35.0 | 0     | 0     | 8.0500  | S        | Third | man   | True       | N  |

# Pivot Tables by Hand

- ▶ To start learning more about this data, we might begin by grouping according to gender, survival status, or some combination thereof
- ▶ For example, let's look at survival rate by gender, using a GroupBy operation:

```
titanic.groupby('sex')['survived'].mean()
```

```
sex
female 0.742038
male 0.188908
Name: survived, dtype: float64
```

- ▶ This is useful, but we might like to go one step deeper and look at survival by both sex and class

# Pivot Tables by Hand

- ▶ Using a GroupBy operation, we might proceed like this:
  - ▶ *We group by class and gender*
  - ▶ *select survival*
  - ▶ *apply a mean aggregate*
  - ▶ *combine the resulting groups*
  - ▶ *unstack the hierarchical index to reveal the hidden multidimensionality*

```
titanic.groupby(['sex', 'class'])['survived'].mean().unstack()
```

|        | class    | First    | Second   | Third |
|--------|----------|----------|----------|-------|
| sex    |          |          |          |       |
| female | 0.968085 | 0.921053 | 0.500000 |       |
| male   | 0.368852 | 0.157407 | 0.135447 |       |

- ▶ This 2D GroupBy is common enough that Pandas includes the function **pivot\_table()**, which succinctly handles this type of multi-dimensional aggregation

# Pivot Table Syntax

- Here is the equivalent to the preceding operation using the `pivot_table()` method of DataFrames:

```
titanic.pivot_table('survived', index='sex', columns='class')
```

| class  | First    | Second   | Third    |
|--------|----------|----------|----------|
| sex    |          |          |          |
| female | 0.968085 | 0.921053 | 0.500000 |
| male   | 0.368852 | 0.157407 | 0.135447 |

- This is definitely more readable than the groupby approach, and produces the same result

# Multi-Level Pivot Tables

- ▶ Just as in the GroupBy, the grouping in pivot tables can be specified with multiple levels, and via a number of options
- ▶ For example, we might be interested in looking at age as a third dimension
- ▶ We'll bin the age using **pd.cut()**, which bins values into discrete intervals:

```
age = pd.cut(titanic['age'], [0, 18, 80])
titanic.pivot_table('survived', ['sex', 'age'], 'class')
```

|        | class    | First    | Second   | Third    |
|--------|----------|----------|----------|----------|
| sex    | age      |          |          |          |
| female | (0, 18]  | 0.909091 | 1.000000 | 0.511628 |
|        | (18, 80] | 0.972973 | 0.900000 | 0.423729 |
| male   | (0, 18]  | 0.800000 | 0.600000 | 0.215686 |
|        | (18, 80] | 0.375000 | 0.071429 | 0.133663 |

# Multi-Level Pivot Tables

- ▶ We can apply the same strategy when working with the columns as well
- ▶ Let's add info on the fare paid using **pd.qcut()** to automatically compute quantiles:

```
fare = pd.qcut(titanic['fare'], 3)
titanic.pivot_table('survived', ['sex', 'age'], [fare, 'class'])
```

|        | fare     | (-0.001, 8.662] |          |       | (8.662, 26.0] |          |          | (26.0, 512.329] |          |  |
|--------|----------|-----------------|----------|-------|---------------|----------|----------|-----------------|----------|--|
|        | class    | First           | Third    | First | Second        | Third    | First    | Second          | Third    |  |
| sex    | age      |                 |          |       |               |          |          |                 |          |  |
| female | (0, 18]  | NaN             | 0.700000 | NaN   | 1.000000      | 0.583333 | 0.909091 | 1.0             | 0.111111 |  |
|        | (18, 80] | NaN             | 0.523810 | 1.0   | 0.877551      | 0.433333 | 0.972222 | 1.0             | 0.125000 |  |
| male   | (0, 18]  | NaN             | 0.166667 | NaN   | 0.500000      | 0.500000 | 0.800000 | 0.8             | 0.052632 |  |
|        | (18, 80] | 0.0             | 0.127389 | 0.0   | 0.086957      | 0.102564 | 0.400000 | 0.0             | 0.500000 |  |

- ▶ The result is a four-dimensional aggregation with hierarchical indices

# Additional Pivot Table Options

- ▶ The **aggfunc** keyword controls what type of aggregation is applied, which is a mean by default
- ▶ As in the GroupBy, the aggregation specification can be a string representing one of several common choices (e.g., 'sum', 'mean', 'count', 'min', 'max', etc.) or a function that implements an aggregation (e.g., np.sum(), min(), sum(), etc.)
- ▶ Also, it can be specified as a dictionary mapping a column to any of the above:

```
titanic.pivot_table(index='sex', columns='class',
 aggfunc={'survived': sum, 'fare': 'mean'})
```

|        | fare       |           |           | survived |        |       |
|--------|------------|-----------|-----------|----------|--------|-------|
| class  | First      | Second    | Third     | First    | Second | Third |
| sex    |            |           |           |          |        |       |
| female | 106.125798 | 21.970121 | 16.118810 | 91       | 70     | 72    |
| male   | 67.226127  | 19.741782 | 12.661633 | 45       | 17     | 47    |

# Additional Pivot Table Options

- ▶ At times it's useful to compute totals along each grouping
- ▶ This can be done via the **margins** keyword:

```
titanic.pivot_table('survived', index='sex', columns='class', margins=True)
```

| class  | First    | Second   | Third    | All      |
|--------|----------|----------|----------|----------|
| sex    |          |          |          |          |
| female | 0.968085 | 0.921053 | 0.500000 | 0.742038 |
| male   | 0.368852 | 0.157407 | 0.135447 | 0.188908 |
| All    | 0.629630 | 0.472826 | 0.242363 | 0.383838 |

- ▶ This automatically gives us information about the class-agnostic survival rate by gender, the gender-agnostic survival rate by class, and the overall survival rate of 38%
- ▶ The margin label can be specified with the `margins_name` keyword, which defaults to "All"

# Time Series

- ▶ Pandas was developed in the context of financial modeling, so as you might expect, it contains a fairly extensive set of tools for working with dates, times, and time-indexed data
- ▶ Date and time data comes in a few flavors:
  - ▶ *Time stamps* reference particular moments in time (e.g., July 4th, 2015 at 7:00am).
  - ▶ *Time intervals* reference a length of time between a particular beginning and end point, for example, the year 2015
  - ▶ *Periods* usually reference a special case of time intervals in which each interval is of uniform length and does not overlap (e.g., 24 hour-long periods comprising days)
  - ▶ *Time deltas* or *durations* reference an exact length of time (e.g., a duration of 22.56 seconds)

# Time Series Data Structures

- ▶ For time stamps, Pandas provides the **Timestamp** type
  - ▶ This is essentially a replacement for Python's native datetime, but is based on the more efficient numpy.datetime64 data type
  - ▶ The associated Index structure is DatetimeIndex
- ▶ For time Periods, Pandas provides the **Period** type
  - ▶ This encodes a fixed-frequency interval based on numpy.datetime64
  - ▶ The associated index structure is PeriodIndex
- ▶ For time deltas or durations, Pandas provides the **Timedelta** type
  - ▶ Timedelta is a more efficient replacement for Python's native datetime.timedelta type, and is based on numpy.timedelta64
  - ▶ The associated index structure is TimedeltaIndex

# Timestamp

- ▶ Pandas provides a **Timestamp** object, which combines the ease-of-use of Python's datetime with the efficient storage and vectorized interface of numpy.datetime64
- ▶ **pd.to\_datetime()** can be used to create a Timestamp from a wide variety of formats:

```
date = pd.to_datetime("4th of July, 2018")
date
```

```
Timestamp('2018-07-04 00:00:00')
```

```
date.strftime('%A')
```

```
'Wednesday'
```

- ▶ We can do NumPy-style vectorized operations directly on this same object:

```
date + pd.to_timedelta(np.arange(12), 'D')
```

```
DatetimeIndex(['2018-07-04', '2018-07-05', '2018-07-06', '2018-07-07',
 '2018-07-08', '2018-07-09', '2018-07-10', '2018-07-11',
 '2018-07-12', '2018-07-13', '2018-07-14', '2018-07-15'],
 dtype='datetime64[ns]', freq=None)
```

# Indexing By Time

- ▶ Where the Pandas time series tools really become useful is when you begin to *index data by timestamps*
- ▶ For example, we can construct a Series object that has time indexed data:

```
index = pd.DatetimeIndex(['2017-07-04', '2017-08-04',
 '2018-07-04', '2018-08-04'])
data = pd.Series([0, 1, 2, 3], index=index)
data
```

```
2017-07-04 0
2017-08-04 1
2018-07-04 2
2018-08-04 3
dtype: int64
```

- ▶ We can now use any of the Series indexing patterns, passing date values:

```
data['2017-07-04':'2018-07-04']
```

```
2017-07-04 0
2017-08-04 1
2018-07-04 2
dtype: int64
```

# Indexing By Time

- ▶ There are additional special date-only indexing operations, such as passing a year to obtain a slice of all data from that year:

```
data['2018']
```

```
2018-07-04 2
2018-08-04 3
dtype: int64
```

- ▶ When **pd.to\_datetime()** is called with a series of dates, it yields a DatetimeIndex:

```
dates = pd.to_datetime([datetime(2018, 7, 3), '4th of July, 2018',
 '2018-Jul-6', '07-07-2018', '20180708'])
```

```
dates
```

```
DatetimeIndex(['2018-07-03', '2018-07-04', '2018-07-06', '2018-07-07',
 '2018-07-08'],
 dtype='datetime64[ns]', freq=None)
```

# Indexing By Time

- ▶ Any DatetimeIndex can be converted to a PeriodIndex with the `to_period()` function with the addition of a frequency code
- ▶ Here we'll use 'D' to indicate daily frequency:

```
dates.to_period('D')

PeriodIndex(['2018-07-03', '2018-07-04', '2018-07-06', '2018-07-07',
 '2018-07-08'],
 dtype='period[D]', freq='D')
```

- ▶ A TimedeltaIndex is created, for example, when a date is subtracted from another:

```
dates - dates[0]

TimedeltaIndex(['0 days', '1 days', '3 days', '4 days', '5 days'],
 dtype='timedelta64[ns]', freq=None)
```

# Regular Sequences

- ▶ To make the creation of regular date sequences more convenient, Pandas offers a few functions for this purpose:
  - ▶ `pd.date_range()` for timestamps
  - ▶ `pd.period_range()` for periods
  - ▶ `pd.timedelta_range()` for time deltas
- ▶ We've seen that Python's `range()` and NumPy's `np.arange()` turn a startpoint, endpoint, and optional stepsize into a sequence
- ▶ Similarly, `pd.date_range()` accepts a start date, an end date, and an optional frequency code to create a regular sequence of dates
- ▶ By default, the frequency is one day:

```
pd.date_range('2018-07-27', '2018-08-03')
```

```
DatetimeIndex(['2018-07-27', '2018-07-28', '2018-07-29', '2018-07-30',
 '2018-07-31', '2018-08-01', '2018-08-02', '2018-08-03'],
 dtype='datetime64[ns]', freq='D')
```

# Regular Sequences

- ▶ Alternatively, the date range can be specified not with a start and endpoint, but with a startpoint and a number of periods:

```
pd.date_range('2018-07-27', periods=8)

DatetimeIndex(['2018-07-27', '2018-07-28', '2018-07-29', '2018-07-30',
 '2018-07-31', '2018-08-01', '2018-08-02', '2018-08-03'],
 dtype='datetime64[ns]', freq='D')
```

- ▶ The spacing can be modified by altering the **freq** argument, which defaults to D
- ▶ For example, here we will construct a range of hourly timestamps:

```
pd.date_range('2018-07-27', periods=8, freq='H')

DatetimeIndex(['2018-07-27 00:00:00', '2018-07-27 01:00:00',
 '2018-07-27 02:00:00', '2018-07-27 03:00:00',
 '2018-07-27 04:00:00', '2018-07-27 05:00:00',
 '2018-07-27 06:00:00', '2018-07-27 07:00:00'],
 dtype='datetime64[ns]', freq='H')
```

# Regular Sequences

- ▶ To create regular sequences of Period or Timedelta values, the very similar **pd.period\_range()** and **pd.timedelta\_range()** functions are useful
- ▶ Here are some monthly periods:

```
pd.period_range('2018-07', periods=8, freq='M')

PeriodIndex(['2018-07', '2018-08', '2018-09', '2018-10', '2018-11', '2018-12',
 '2019-01', '2019-02'],
 dtype='period[M]', freq='M')
```

- ▶ And a sequence of durations increasing by an hour:

```
pd.timedelta_range(0, periods=10, freq='H')

TimedeltaIndex(['00:00:00', '01:00:00', '02:00:00', '03:00:00', '04:00:00',
 '05:00:00', '06:00:00', '07:00:00', '08:00:00', '09:00:00'],
 dtype='timedelta64[ns]', freq='H')
```

# Frequencies and Offsets

- ▶ Just as we saw the D (day) and H (hour) codes above, we can use such codes to specify any desired frequency spacing
- ▶ The following table summarizes the main codes available:

| Code | Description  |
|------|--------------|
| D    | Calendar day |
| W    | Weekly       |
| M    | Month end    |
| Q    | Quarter end  |
| A    | Year end     |
| H    | Hours        |
| T    | Minutes      |
| S    | Seconds      |
| L    | Milliseconds |
| U    | Microseconds |

| Code | Description          |
|------|----------------------|
| B    | Business day         |
| BM   | Business month end   |
| BQ   | Business quarter end |
| BA   | Business year end    |
| BH   | Business hours       |

# Frequencies and Offsets

- ▶ The monthly, quarterly, and annual frequencies are all marked at the end of the specified period
- ▶ By adding an **S** suffix to any of these, they instead will be marked at the beginning:

| Code | Description   |
|------|---------------|
| MS   | Month start   |
| QS   | Quarter start |
| YS   | Year start    |

| Code | Description            |
|------|------------------------|
| BMS  | Business month start   |
| BQS  | Business quarter start |
| BAS  | Business year start    |

- ▶ Additionally, you can change the month used to mark any quarterly or annual code by adding a three-letter month code as a suffix:
  - ▶ Q-JAN, BQ-FEB, QS-MAR, BQS-APR, etc.
  - ▶ A-JAN, BA-FEB, AS-MAR, BAS-APR, etc.
- ▶ In the same way, the split-point of the weekly frequency can be modified by adding a three-letter weekday code: W-SUN, W-MON, W-TUE, W-WED, etc.

# Frequencies and Offsets

- ▶ On top of this, codes can be combined with numbers to specify other frequencies
- ▶ For example, for a frequency of 2 hours 30 minutes, we can combine the hour (H) and minute (T) codes as follows:

```
pd.timedelta_range(0, periods=9, freq='2H30T')

TimedeltaIndex(['00:00:00', '02:30:00', '05:00:00', '07:30:00', '10:00:00',
 '12:30:00', '15:00:00', '17:30:00', '20:00:00'],
 dtype='timedelta64[ns]', freq='150T')
```

# Resampling, Shifting and Windowing

- ▶ The ability to use dates and times as indices provides the benefits of indexed data in general (e.g., automatic alignment during operations, intuitive data slicing and access, etc.), and Pandas provides several additional time series-specific operations
- ▶ We will take a look at a few of those, using Google's stock closing price history:

```
from pandas_datareader import data

goog = data.DataReader('GOOG', start='2014', end='2018',
 data_source='iex')
goog.head()
```

5y

|            | open    | high   | low    | close  | volume |
|------------|---------|--------|--------|--------|--------|
| date       |         |        |        |        |        |
| 2014-03-27 | 568.000 | 568.00 | 552.92 | 558.46 | 13052  |
| 2014-03-28 | 561.200 | 566.43 | 558.67 | 559.99 | 41003  |
| 2014-03-31 | 566.890 | 567.00 | 556.93 | 556.97 | 10772  |
| 2014-04-01 | 558.710 | 568.45 | 558.71 | 567.16 | 7932   |
| 2014-04-02 | 565.106 | 604.83 | 562.19 | 567.00 | 146697 |

# Resampling, Shifting and Windowing

- ▶ For simplicity, we'll use just the closing price:

```
goog = goog['close']
```

- ▶ We will also convert the index into a datetime index:

```
Converting the index into a datetime index
goog.index = pd.to_datetime(goog.index)
```

- ▶ We can visualize this using plot(), after the normal Matplotlib setup boilerplate:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn
seaborn.set()
```

```
goog.plot();
```



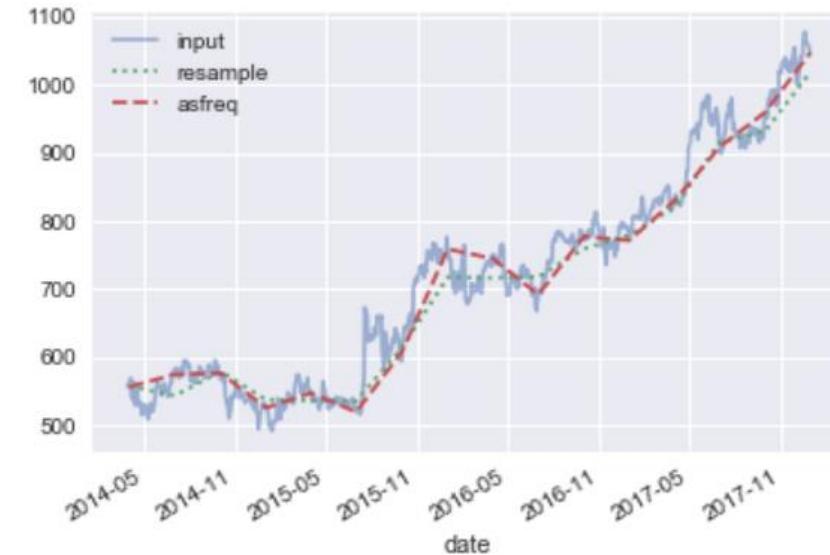
# Resampling

- ▶ One common need for time series data is resampling at a higher or lower frequency
- ▶ This can be done using the **resample()** method, or the much simpler **asfreq()** method
- ▶ The primary difference between the two is that **resample()** is fundamentally a *data aggregation*, while **asfreq()** is fundamentally a *data selection*
- ▶ For example, let's down-sample the data at the end of business quarters:

```
goog.plot(alpha=0.5, style='--')

report the average of the previous quarter
goog.resample('BQ').mean().plot(style=':')

report the value at the end of the quarter
goog.asfreq('BQ').plot(style='--')
plt.legend(['input', 'resample', 'asfreq']);
```



# Resampling

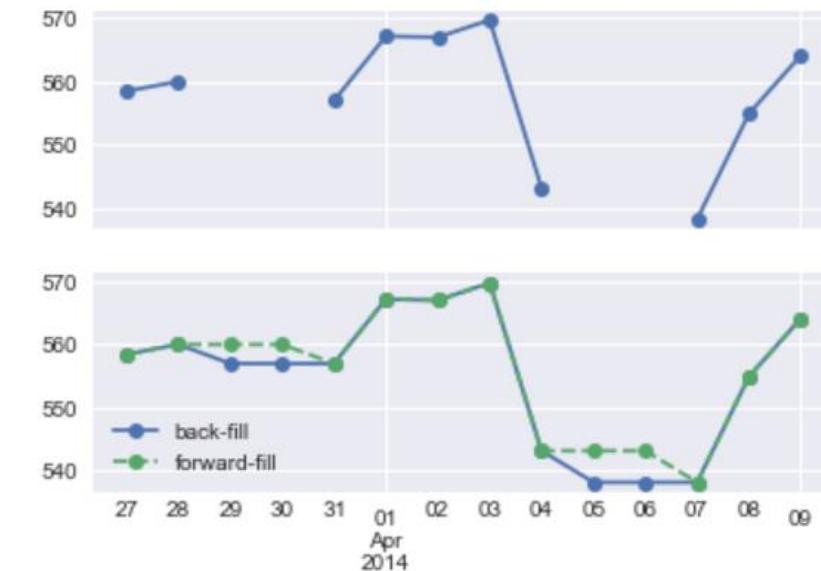
- ▶ For up-sampling, resample() and asfreq() are largely equivalent, though resample has many more options available
- ▶ The default for both methods is to fill the up-sampled points with NA values
- ▶ However, they accept a **method** argument to specify how values are imputed
- ▶ For example, let's resample the stock data at a daily frequency (including weekends):

```
fig, ax = plt.subplots(2, sharex=True)
data = goog.iloc[:10]

data.asfreq('D').plot(ax=ax[0], marker='o')

data.asfreq('D', method='bfill').plot(ax=ax[1], style='--o')
data.asfreq('D', method='ffill').plot(ax=ax[1], style='--o')

ax[1].legend(["back-fill", "forward-fill"]);
```



## Time Shifts

- ▶ Another common time series-specific operation is shifting of data in time
- ▶ Pandas has two closely related methods for computing this: **shift()** and **tshift()**
- ▶ The difference is that `shift()` *shifts the data*, while `tshift()` *shifts the index*
- ▶ In both cases, the shift is specified in multiples of `mdates.DateFormatter` the frequency
- ▶ Here we will both `shift()` and `tshift()` by 400 days:

# Time Shifts

```

fig, ax = plt.subplots(3, figsize=(7, 10))

apply a frequency to the data
goog = goog.asfreq('D', method='ffill')

goog.plot(ax=ax[0])
goog.shift(400).plot(ax=ax[1])
goog.tshift(400).plot(ax=ax[2])

Legends and annotations
local_max = pd.to_datetime('2015-7-17')
offset = pd.Timedelta(400, 'D')

ax[0].legend(['input'], loc=2)
ax[0].axvline(local_max, alpha=0.3, color='red')
set monthly locator

ax[1].legend(['shift(400)'], loc=2)
ax[1].axvline(local_max + offset, alpha=0.3, color='red')

ax[2].legend(['tshift(400)'], loc=2)
ax[2].axvline(local_max + offset, alpha=0.3, color='red');

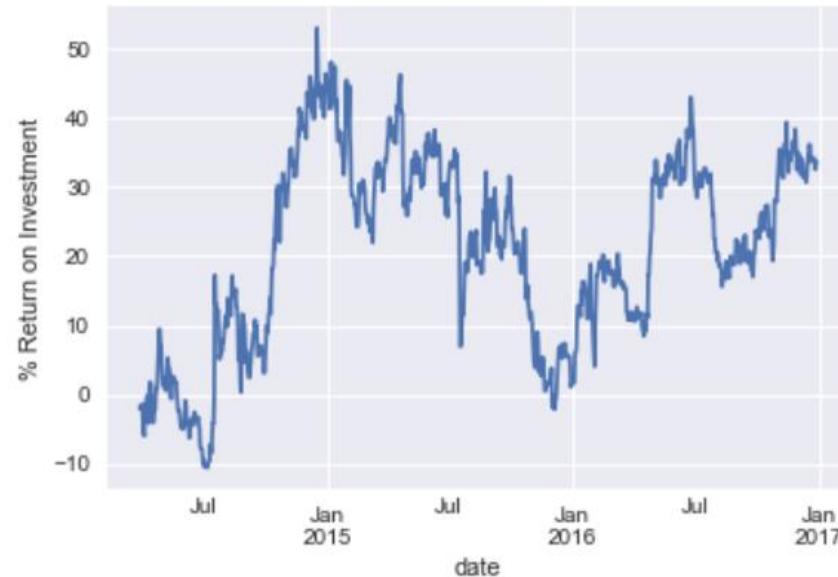
```



# Time Shifts

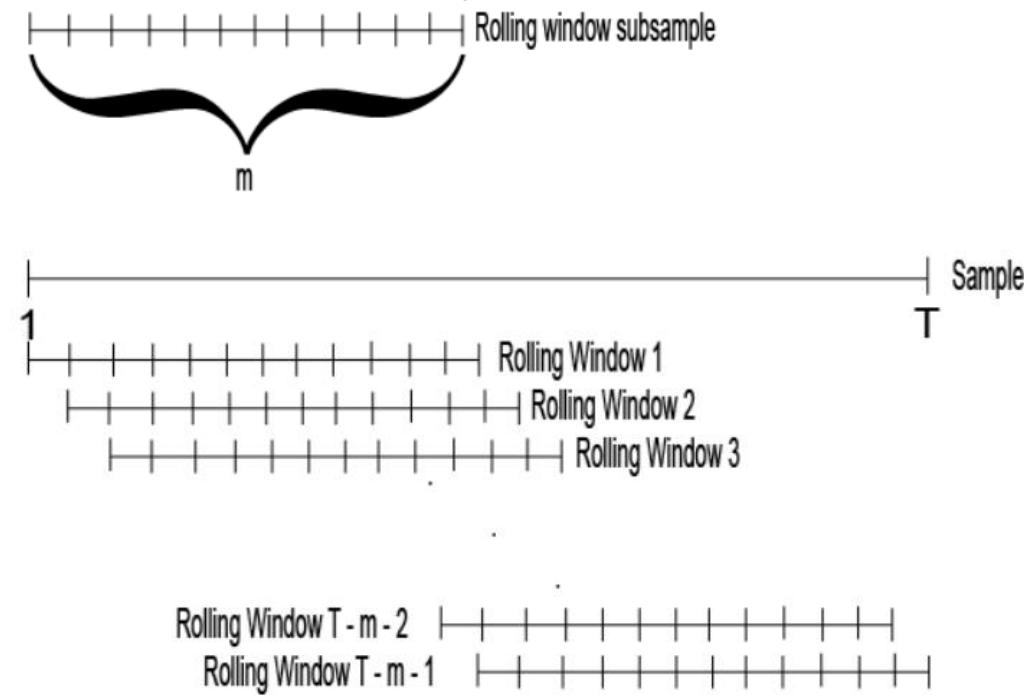
- ▶ A common context for this type of shift is in computing differences over time
- ▶ For example, we can use shifted values to compute the one-year return on investment for Google stock over the course of the dataset:

```
ROI = 100 * (goog.tshift(-365) / goog - 1)
ROI.plot()
plt.ylabel('% Return on Investment');
```



# Rolling Windows

- In rolling windows analysis, we divide the data set into overlapping sub-samples, and calculate the aggregated values by moving the window forward one observation at a time:



# Rolling Windows

- ▶ The **rolling()** method of Series and DataFrame provides rolling window calculations:

```
ser = pd.Series(np.arange(10, 16))
ser
```

```
0 10
1 11
2 12
3 13
4 14
5 15
dtype: int32
```

```
Rolling sum for a 3 rows window
ser.rolling(3).sum()
```

```
0 NaN
1 NaN
2 33.0
3 36.0
4 39.0
5 42.0
dtype: float64
```

- ▶ As with group-by operations, the aggregate() and apply() methods can be used for custom rolling computations

# Rolling Windows

- ▶ Rolling-window analysis of a time-series model assesses:
  - ▶ The stability of the model over time, by examining whether the coefficients of the model are time-invariant
  - ▶ The forecast accuracy of the model
- ▶ For example, here is the one-year centered rolling mean and standard deviation of the Google stock prices:

```
rolling = goog.rolling(365, center=True)
data = pd.DataFrame({'input': goog,
 'one-year rolling_mean': rolling.mean(),
 'one-year rolling_std': rolling.std()})
ax = data.plot(style=['-', '--', ':']);
ax.lines[0].set_alpha(0.5)
```

