Python Fundamentals

MathByte Academy

Course Overview

1

What this course is about…

→ introduction to Python from first principles

→ you will have the knowledge and understanding to learn more on your own

→ get insights into 3rd party Python libraries in general

→ learn the basics of some common 3rd Party libraries

→ numerical computations

→ manipulating data sets

→ charting

Content Overview

→ installing and running Python

→ basic principles

numerical types (integers, floats, booleans)
variables
operators (arithmetic, logical, boolean)

→ control flow

conditional execution
iteration (iterables, iterators, loops)
exception handling

→ advanced data types

sequence types (lists, tuples, strings)
dictionaries and sets
dates and times
decimals

Content Overview

→ functional programming

functions
higher-order functions
closures
decorators

→ object-oriented programming

custom classes
methods
properties

→ data acquisition

CSV
JSON
Using REST APIs

Content Overview

→ 3rd party Libraries

pytz

dateutil

(http) requests

numpy

pandas

matplotlib

Prerequisites

→ Windows, Mac

→ needs to run Python 3.6+ (recommend at least 3.8/3.9 or higher)

Windows 10 Mac 10.9 and higher

→ Linux – you'll need to find/use installation instructions

→ some familiarity with Terminal (Mac/Linux) / Command Prompt (Windows)

→ will be needed to install and run Python

→ just the basics

→ how to open/terminate a shell

→ change directory

→ list contents of a directory

→ create a directory

→ no prior Python knowledge needed

→ prior programming knowledge helpful, but not required

Course Structure and Materials

→ each topic is organized in two parts

→ lecture video sit back , watch, take notes if you like

→ coding video lean in and code along – pause video, rewind, type code!

→ exercises

→ each section (except installation section) has a set of exercises

→ make sure you are confident before moving on to the next section

→ downloadable course materials

→ all coding videos have an accompanying Jupyter Notebook / resources

→ contains all the code we do in the code video

→ contains any required extras (such as data sets)

→ notebooks are fully annotated

Installing and Running Python

2

→ what is Python?

→ language vs implementation

→ the canonical, or reference implementation of Python

→ installing Python

→ side by side versions of Python

→ virtual environments

→ installing 3rd party libraries

→ running Python code

→ interactive mode

→ script mode

Python

A bit of history…

→ created by Guido van Rossum in 1989 while working at CWI
(National Research Institute for Mathematics and Computer Science, Netherlands)

→ became a community driven effort, overseen by Guido

→ who became the BDFL (benevolent dictator for life) (stepped down in 2018)

→ many developers ("core" developers) have contributed over the years

→ was named after the British comedy group Monty Python
(who says developers don't have a sense of humour!)

→ still actively developed today

Python 2.0 released in 2000 → last release was 2.7 (end of life April 2020)

Python 3.0 released in 2008 → 3.9 released in October 2020

What is Python?

→ Python is a language, not an application

→ there are many implementations of Python

CPython

PyPy

→ even compilers that "translate" Python code to other languages

IronPython → .NET

Jython → Java

Cython → C/C++

→ the "reference" Python implementation is CPython

CPython

→ reference implementation → https://www.python.org

→ most widely used distribution of Python

→ open source → written in C https://github.com/python/cpython

→ includes the standard library

→ a collection of additional functionality that goes beyond just the Python language

→ written in C and Python

→ this implementation of Python and the standard library is the "official" implementation

→ many platforms Linux, Windows, Mac OS, iOS, Android, PlayStation, Xbox,…

Installing Python

Installing CPython

→ CPython is basically a bunch of files, located in some directory on your computer

→ one of those files is an executable that is used to run Python code files or an interactive shell

→ entire standard library is also included in these files

→ to "install" CPython you simply copy these files into a directory

→ possible to have multiple versions of CPython on the same computer

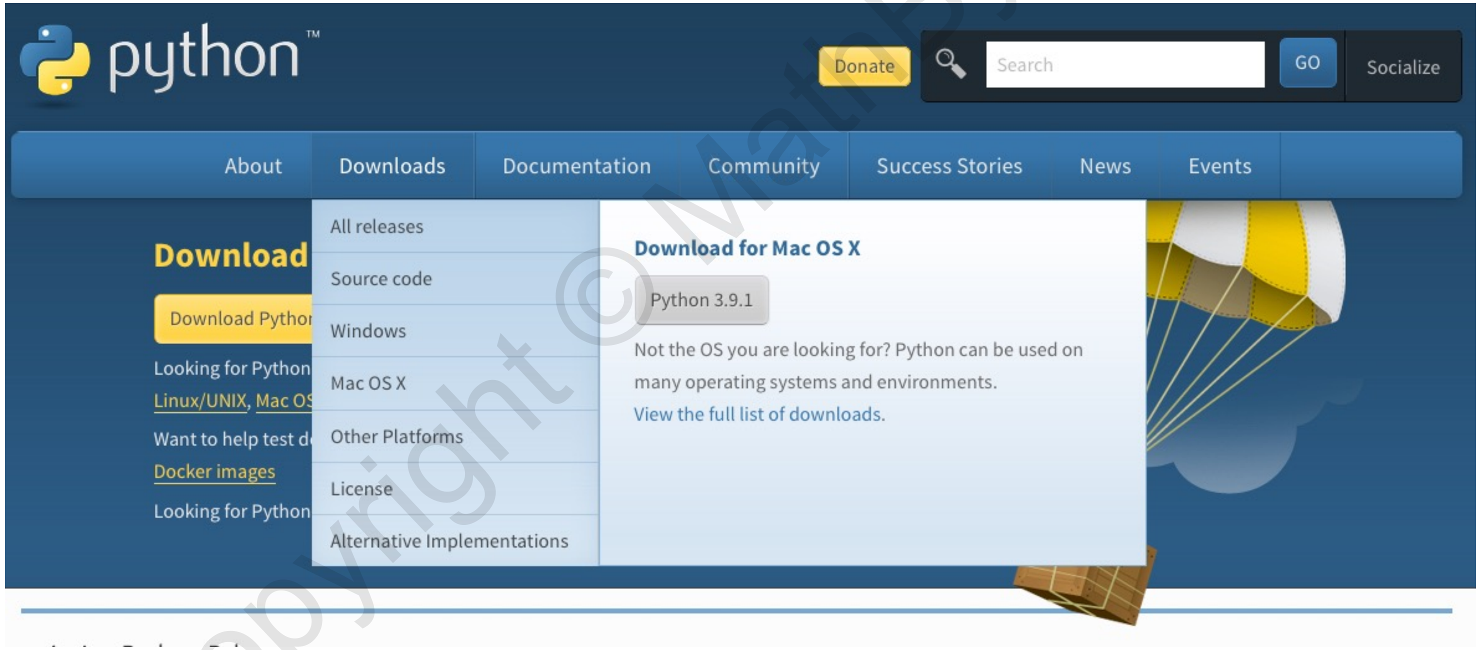
→ just install the files in different directories

→ call the desired Python executable

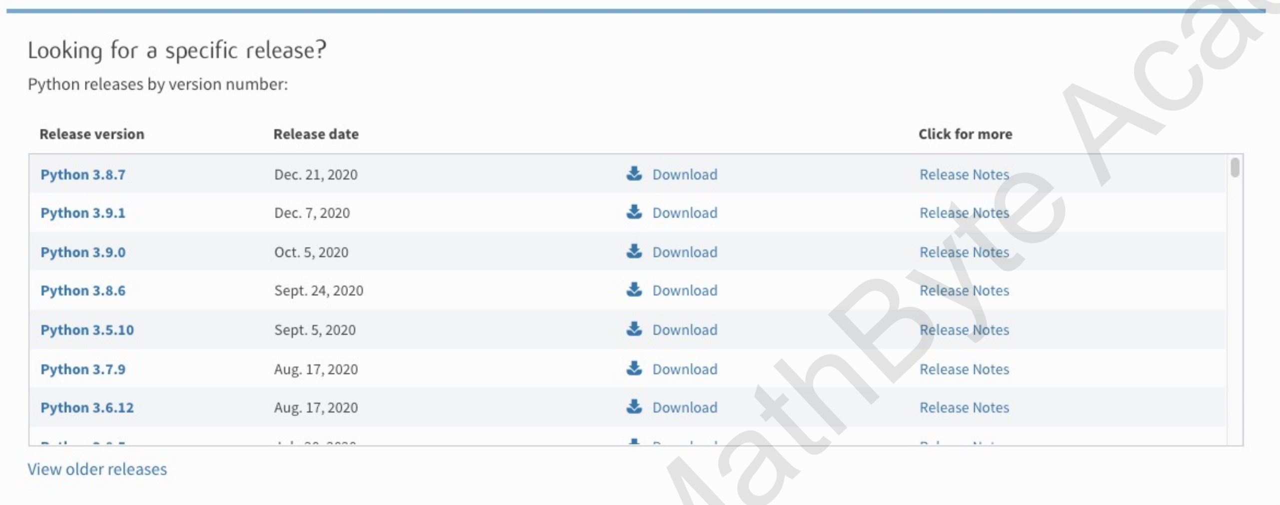
Where to find installation packages

→ https://www.python.org

→ click on Downloads → all releases



→ choose which version you want and what OS you are on



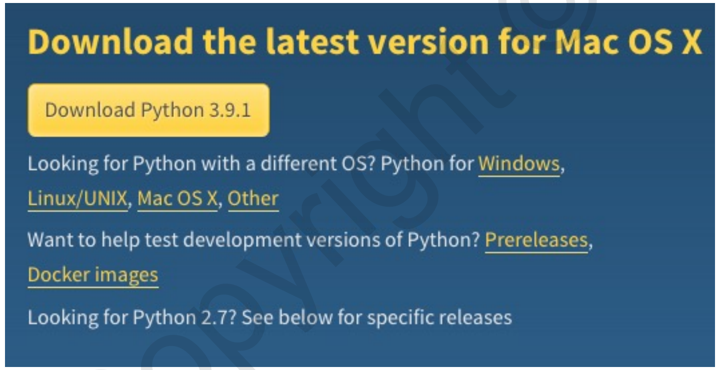
Looking for a specific release?

Python releases by version number:

Release version	Release date		Click for more
Python 3.8.7	Dec. 21, 2020	Download	Release Notes
Python 3.9.1	Dec. 7, 2020	Download	Release Notes
Python 3.9.0	Oct. 5, 2020	Download	Release Notes
Python 3.8.6	Sept. 24, 2020	Download	Release Notes
Python 3.5.10	Sept. 5, 2020	Download	Release Notes
Python 3.7.9	Aug. 17, 2020	Download	Release Notes
Python 3.6.12	Aug. 17, 2020	Download	Release Notes

View older releases

→ or use default (which should recognize your OS)



→ two videos included in this course

→ Windows Installation

→ Mac Installation

(Linux installation is same as Mac, just download for Linux OS)

→ jump directly to your specific platform video

use Python 3.8.1 or higher for this course

Virtual Environments

3^{rd} Party Libraries

→ many 3^{rd} party libraries exist

→ add-ons to Python for more specialized functionality

→ a bunch of files

→ that get added to your Python "installation"

→ 3^{rd} party libraries often rely on other 3^{rd} party libraries

→ or might have specific releases for specific Python versions

→ this can create conflicts!

my_app_1 → some_lib_1.0 (breaks with some_lib_1.1 and higher)

my_app_2 → some_lib_1.1 (breaks with some_lib_1.0 and lower)

Solution

→ since Python is just a directory of files

→ create two copies of this directory

/usr/user1/python3.9-ENV1/ /usr/user1/python3.9-ENV2/

install some_lib_1.0 in here install some_lib_1.1 in here

→ run your apps/shell using the appropriate Python directory

→ /Users/user1/python3.9-ENV1/bin/python my_app.py

→ /Users/user1/python3.9-ENV2/bin/python my_other_app.py

→ typing this long path is tedious

→ add to PATH environment variable (tells OS where to look for executables)

Solution

/usr/user1/python3.9-ENV1/

→ add /usr/user1/python3.9-ENV1/ to (front of) PATH

→ python my_app.py

/usr/user1/python3.9-ENV2/

→ remove /usr/user1/python3.9-ENV1/ from path

→ add /usr/user1/python3.9-ENV2/ to (front of) PATH

→ python my_other_app.py

→ works but can become tedious as well!

Virtual Environments

→ these are used to perform the exact same steps

→ make copy of Python installation

→ provides scripts to "activate"/"deactivate" the environment

→ unsets old path / sets new path

→ very efficient on Unix/Mac – uses symlinks

→ little less efficient on Windows – actually copies the files

→ provides solution to version conflicts

→ use them!!

Creating Virtual Environments

→ different implementations of this have evolved over the years

→ Python now has a virtual environment manager built-in

→ we'll use that one in this course

→ first decide which Python version to use

→ Mac: python3.8, python3.9, etc

→ Windows: specify full path to python version

```
<python version/path> -m venv <your_env_name>
```

→ creates a new virtual environment

Activating the Virtual Environment

→ activating a virtual environment essentially modifies your PATH

→ when you type python on command line after activating environment

→ you will be running version of Python located in that environment directory

→ different for Mac/Linux vs Windows

Mac/Linux: `source <path_to_env>/bin/activate`

Windows: `<path_to_env>\Scripts\activate.bat`

Coding

Installing Packages

`pip`: Package Installer for Python

→ installing 3rd party libraries (packages) basically requires copying files into the Python installation (whichever directory you want)

→ can be done manually, but again, tedious

→ instead can use another app that is installed alongside Python

→ `pip`

→ easily install, update and remove packages

→ uses the Python Package Index https://pypi.org

→ official 3rd party repository for Python

→ a repository of over 200,000 Python packages!

Installing Packages with `pip`

→ activate the virtual environment first (sets your PATH)

→ `pip install package_name`

→ can even specify versions

```
pip install package_name==1.3.2

pip install package_name<=1.2

pip install package_name>2.0
```

→ once you have decided on a specific version of a package for your project you should always use same version number if creating a new environment for the same project

→ otherwise you could end up breaking your own code!

The `requirements.txt` File

→ use a file, alongside your code, to keep track of required packages and versions

```
requirements.txt
numpy==1.18.1
pandas==1.1.4
matplotlib==3.3.3
```

→ file name can be anything

→ requirements.txt is a standard convention

```
> pip install -r requirements.txt
```

→ easier (install everything with one command)

→ reproducibility / consistency

→ documentation

→ for this course a `requirements.txt` file is available in your downloads

→ defines (and pins) all specific libraries we'll need

→ please use it so we're all using the same library versions

(functionality can change from version to version)

→ will install many libraries (and their dependencies)

`requests`

`pytz`

`numpy`

`pandas` and more…

Summary of Steps

→ create a new virtual environment

→ activate the environment

→ pip install libraries (aka packages)

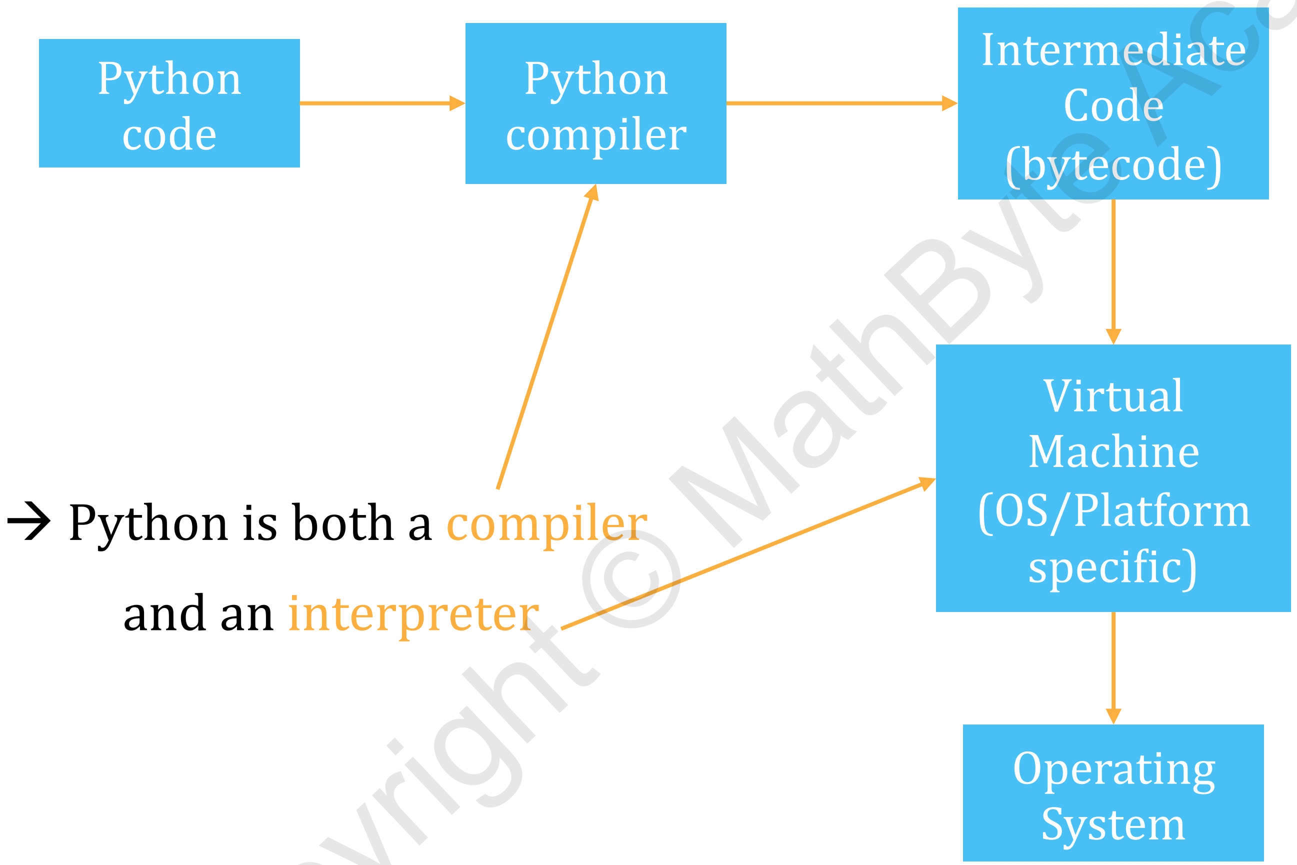
→ don't forget to activate the environment before you pip install!

→ otherwise pip install will install these packages to your reference Python install

Coding

Running Python

Python Compiler/Interpreter



How do we "run" Python?

→ Python is a compiler/interpreter

→ reads in a chunk of code (your program)

→ compiles and runs it

→ output is sent to your screen (console)

→ Python can do this in two ways

→ interactive mode

→ you type a Python line/block of code and execute it immediately

→ any output is immediately displayed

→ continue typing/running code one line/block at a time

→ REPL (read-eval-print-loop)

→ script mode

→ write all your code in files first

→ then execute all this code using command line

Interactive Mode

→ activate virtual environment

→ start Python shell (REPL) by typing `python` on command line

  → start typing Python commands

  → non graphical interface

    → perfect when working on GUI-less servers

    → little tedious to use when you are just trying things out

→ Jupyter Notebooks

  → browser based REPL (needs to be pip installed)

  → much easier/nicer to use than command line

  → can save your projects into a file (a notebook)

    → usually `.ipynb` extension

Script Mode

→ write all your code using a text editor

→ run your Python program using command line

```
> python my_app.py
```

→ traditional programs are great for

- → running the same program over and over again
- → better structure
- → complex applications (web server, prediction server, libraries, …)

Python IDEs

IDE → integrated development environment

→ a text editor

→ with many extras for easily running code, debugging, and more

→ runs code using the same command line approach for scripts

→ many popular IDEs / editors around

→ PyCharm

→ VSCode

→ Atom

→ Sublime Text 3

→ Spyder

this is the one I personally use for development

and more…

Coding

Python Basics

3

- → some basic Python types
  - → integers → floats → booleans
- → a brief explanation of what objects are
- → basic operators
  - → arithmetic operators
  - → integer division and modulus
  - → comparison operators
  - → Boolean operators
  - → operator precedence

Basic Data Types

Types

→ Entities in a program always have an associated type

→ in the real world too!

John is a person

My local pharmacy is a store

My bank balance is a (real) number

The number of pages in a book is an (integer) number

The file `budget.xlsx` is an Excel spreadsheet

Statements can be True or False

they have a type → Boolean type (`True`, `False`)

"I am a Python dev" is True

"My dog likes cats" is False

Integers

→ the `int` type

→ used to represent integral numbers: `0`, `1`, `100`, `-100`, etc.

→ integers have an exact representation in Python

→ integers can be of any magnitude

(as long as you have enough memory!)

→ integer numbers can be created from a literal in the Python code

→ `100`

→ `-100`

note how you can use underscores for readability

→ `10_500_000`

→ or, as the result of some calculation (expression)

→ `1 + 1`

Floats

→ the `float` type

→ used to represent real numbers (floating point): `3.14`, `-1.3`

→ can use Python literals to define a `float`

→ `3.14`

→ `-1.3`

→ `1_234.567_876`

→ the **decimal point** differentiates a `float` from an `int` when using literals

`1` → `int`

`1.0` → `float`

Float Representations

Consider the decimal system: `1.234`

In the decimal (base 10) system, this is representable (exactly) using powers of 10:

$$1 + \frac{2}{10} + \frac{3}{100} + \frac{4}{1000}$$

But not all real numbers have a finite representation $\frac{1}{3}$

as a fraction this is exact → but not using a decimal representation

$$\frac{1}{3} = 0.33\dot{3} = \frac{3}{10} + \frac{3}{100} + \frac{3}{1000} + \cdots$$

→ infinite number of fractions

Integer Representations

→ computers only "know" two numbers

→ `0` and `1` → binary system, aka base 2

→ any number in a computer is represented using powers of 2

the binary number `1011`

can be converted to a decimal number:

$$1\times2^0 + 1\times2^1 + 0\times2^2 + 1\times2^3$$
$$= 1 + 2 + 0 + 8$$
$$= 11$$

Float Binary Representation

in the same way, floats are represented using powers of 2 and fractions of powers of 2

$$\frac{1}{2^1} + \frac{0}{2^2} + \frac{1}{2^3} \qquad = \frac{1}{2} + \frac{0}{4} + \frac{1}{8}$$

$$= 0.5 + 0 + 0.125 = 0.625$$

→ we saw that certain numbers do not have a finite decimal representation ($\frac{1}{3}$)

→ same happens with binary representations!

$$0.1 = \frac{0}{2} \quad + \quad \underbrace{\underbrace{\frac{0}{4} + \frac{0}{8} + \frac{1}{16} + \frac{1}{32}}_{0.09375} \quad + \quad \frac{0}{64} + \frac{0}{128} + \frac{1}{256} + \frac{1}{512}}_{0.099609375} + \cdots$$

Floats are not always exact

→ bottom line: not all exact decimal numbers have an exact float representation

→ not a limitation of Python

→ all languages (incl. *Excel*) that use these binary fractions have that issue

⚠ be careful when comparing floats to one another

→ there is a data type that can handle exact representations of decimal fractions

→ `Decimal` (we will look at this type later)

⚠ calculations using `Decimal` are much slower than `float`

Coding

Objects

What are objects?

→ entities created by Python

→ they have state (data)

→ they have methods (functionality)

encapsulation

→ they often represent real world things

Car

State (data)

- brand → Toyota
- model → Prius LE
- # doors → 4
- model_year → 2020
- odometer → 5_402

Functionality

- accelerate( )
- brake( )
- set_cruise_control( )
- left_turn_signal_on( )

Integers are Objects

→ an `int` is an object

→ it has state – the value of the integer

→ but it also has functionality

→ knows how to add itself to another integer

`(10).__add__(100)` → `110`

→ an integer object has the method `__add__` used to implement addition

(this is not how we typically add two integers together – but that's just syntax)

→ knows how to represent itself as a string (e.g. for visual output)

Floats are Objects

→ `float` numbers are objects too

state → value

functionality → `__add__`

other functionality too, for example:

```
(0.125).as_integer_ratio()  → 1, 8
```

Everything in Python is an object

→ any data type we have in Python is actually an object

→ it has state

→ it has functionality

attributes

attributes encompass state and functionality

some attributes are for state

some attributes are for functionality

Dot Notation

If an object has attributes, how do we access those attributes?

→ **dot notation**

`car.brand` → accesses the `brand` attribute of the `car` object

`car.model` → accesses the `model` attribute of the `car` object

For attributes that represent **functionality**, we usually have to **call** the attribute to **perform the action**

→ often supplying additional parameters

```
car.accelerate(10, "mph")

(10).__add__(100)
```

Mutability and Immutability

→ an object is mutable if its internal state can be changed

→ one or more data attributes can be changed

→ an object is immutable if its internal state cannot be changed

→ the state of the object is "set in stone"

In Python many data types are immutable:

- integers
- floats
- booleans
- strings
- …

While some are mutable:

- lists
- dictionaries
- sets
- …

→ we'll cover all these in this course

Coding

Variables

Naming Objects

We often need to label objects with some name

→ reminds us what the object is used for

`apy`   `account_balance`

→ allows us to use the same object in multiple parts of our code

Assigning Names

To assign a label to an object we use the assignment operator **=**

```
account_balance = 1000.0
apy = 0.25
```

this is not the mathematical equality symbol

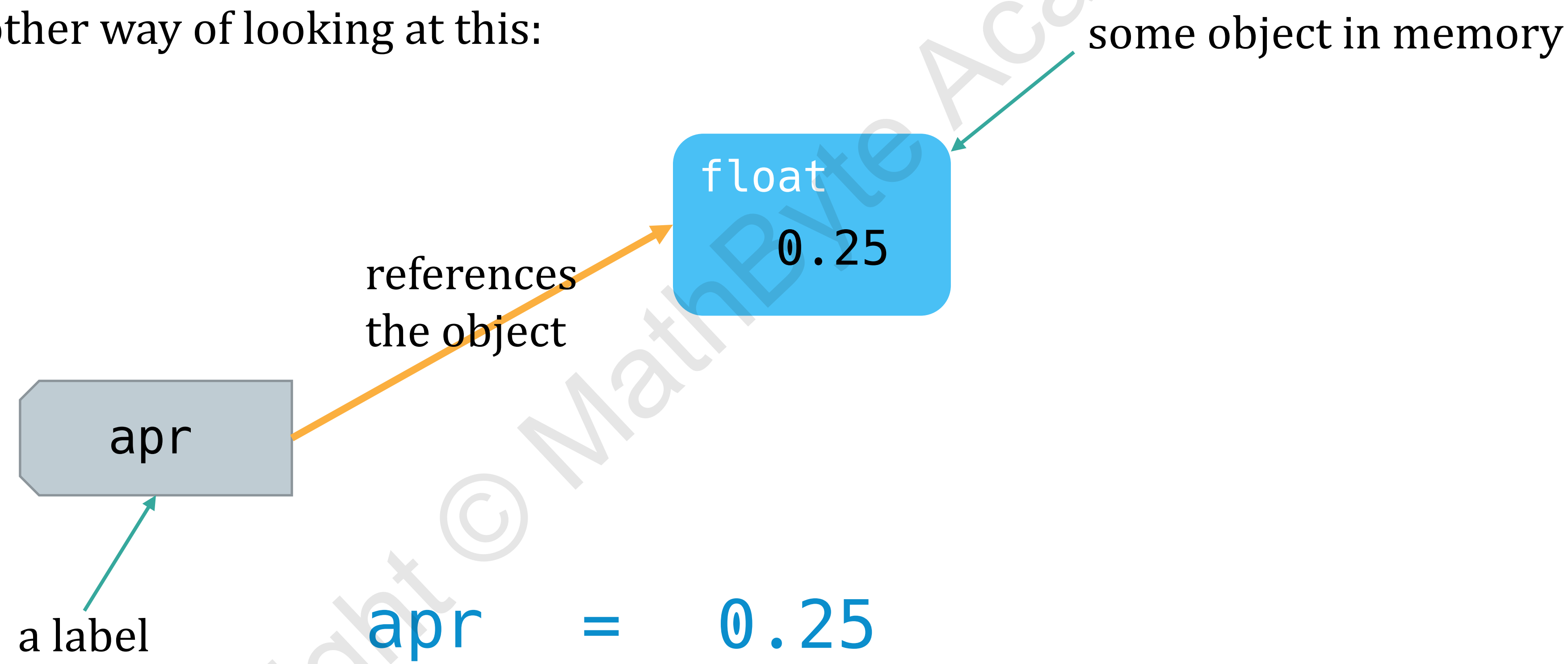
→ we are assigning the label `apy` to the object `0.25`

→ we say `apy` is a **reference** to the float object `0.25`

→ the symbol `apy` is just a label currently pointing to (or referencing) the object `0.25`

References and Variables

Another way of looking at this:




```
apr = 0.25
```

→ `apr` is called a variable

→ but it is just a label (a symbol) that references some object in memory

Variables

So why the term variable?

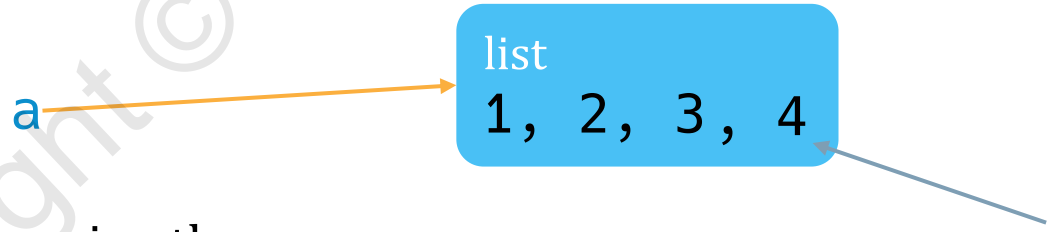
→ over time, which object a symbol references can change

`a = 100` → `a` is referencing the object `100`

later in the program…

`a = True` → `a` is now referencing the object `True`

→ the state of the object the symbol references can change (mutate)



→ `a` is still referencing the same object, but the object's state has changed (mutated)

we append an element to the list

How Variable Assignment Happens

$$\underbrace{\texttt{apy}}_{\text{LHS}} = \underbrace{\texttt{0.25}}_{\text{RHS}}$$

→ Python evaluates the RHS first

→ then it "assigns" that result to the symbol in the LHS

(the LHS becomes a named reference to whatever results from the RHS)

Generally, RHS could be a more complex expression than just a literal

```
balance = 1000.0 - 50.0

circ = 2 * 3.14 * 1.5
```

→ in both cases, the RHS is fully evaluated <u>first</u>

Using Variables

Once a variable has been created, it can be used elsewhere in the program

```
pi = 3.1415

radius = 1

circ = 2 * pi * radius
```

`pi` → 3.1415

`radius` → 1

`circ` → 6.283

→ `circ` is now a reference to the float `6.283`

```
radius = 2
```

`radius` → 2

BUT this does not change `circ`

→ it still points to `6.283`

`circ` → 6.283

Variable Naming

→ case sensitive `apr` is a different symbol than `APR`

→ **must** follow certain rules

→ **should** follow certain conventions

Must-Follow Rules

start with underscore (`_`) or letter (`a-z A-Z`)

(unicode characters are actually OK, but stick to `a-z A-Z`)

followed by any number of underscores or letters, or digits (`0-9`)

```
var     my_var     index1     index_1
_var    __var      __add__
```

all legal

→ cannot be reserved words

```
True  False  if  def  and  or
```

and many more we'll come across in this course

Should-Follow Conventions

PEP 8 Style Guide → typical conventions followed by most Python devs

https://www.python.org/dev/peps/pep-0008/

terminology:

camel case →separate words are distinguished by upper case letters

`accountBalance` `BankAccount`

snake case → separate words are distinguished by underscores

`account_balance` `bank_account`

Should-Follow Conventions

For standard variables:

→ snake case

→ all lower case letters

`account_balance` ✅

`account_Balance` ❌

We'll see other conventions for other special types of objects throughout this course

Should-Follow Conventions

→ Good idea to follow standard conventions

→ but sometimes you may want to break those conventions

→ that's OK – just have a good reason, and be consistent

From the PEP 8 Style Guide:

A foolish consistency is the hobgoblin of little minds.
(Emerson)

Coding

Arithmetic Operators

Terminology

An operator is a programming language symbol that performs some operation on one or more values

Certain types of operators include:

→ arithmetic operators

→ comparison (or relational) operators

→ logical operators

The values an operator acts on are called operands

→ An operator that works on a single operand is called a unary operator

→ An operator that works on two operands is called a binary operator

→ An operator that works on three operands is called a ternary operator

Arithmetic Operators

Unary Operators

`-`	Unary Minus	`-10`
`+`	Unary Plus	`+10`

Binary Operators

`+`	Addition	`10 + 20`
`-`	Subtraction	`20 - 10`
`*`	Multiplication	`10 * 2`
`/`	Division	`10 / 2`
`**`	Power (exponentiation)	`2 ** 4`

→ use parentheses `(` and `)` to group expressions

Operand Types

Arithmetic operators can act on any numerical type

`int` `float`

→ as well as other types we'll encounter later

→ what the operator does is actually determined by the type of the operands

→ an operator *may* support mixed operand types

`2 + 2` → returns an `int`

`2 + 2.0` → returns a `float`

`5.5 * 2` → returns a `float`

`4 / 2` → also returns a `float`!

The Power Operator

The power operator works just like its mathematical counterpart

```
2 ** 4
    → 2 * 2 * 2 * 2
    → 16 (int)
```

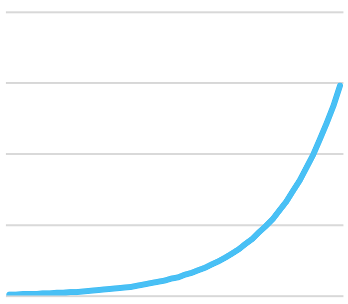
Recall from math: $2^{-a} = \frac{1}{2^a}$

```
2 ** (-4)
    → 1 / (2 ** 4)
    → 1 / 16
    → 0.0625 (float)
```

The Power Operator

→ Python supports floats for either operand of the `**` operator

→ just like mathematical exponentiation



→ graph of $f(x) = e^x$

→ $b^x := e^{x \log(b)}$

→ Python also supports negative bases with real exponents

→ complex numbers

→ it's actually a numerical type in Python (`complex`)

How Python Implements Arithmetic Operators

→ recall: numbers are actually objects

  → they have state
  → they also have functionality

  → one of these is the `__add__` method (amongst many others)

when we do this: `a + b` where `a = 10` and `b = 20`

  → `10` is an `int` object that implements the `__add__` method

  Python actually does this to evaluate the expression:

    → `a.__add__(b)`

→ this works the same way with other types

Looking ahead…

→ any type can choose to implement `__add__` however it wants

→ Python will then use that method to evaluate `type_1 + type_2`

→ we will see later how to create our own types

→ we can implement `__add__` to define `+` for our custom type

→ we'll look at this in code, though some of the code may not make sense (yet!)

Coding

Operator Precedence

When we write an expression such as this: `2 * 10 + 5`

→ what does it mean?

`(2 * 10) + 5` → `25` ? or `2 * (10 + 5)` → `30` ?

Python chooses this

why?

→ operator precedence

Operators have precedence

→ an operator with higher precedence will bind more tightly

→ fancy way of saying it will be evaluated first

Precedence order with arithmetic operators

lower

binary `+  -` (equal precedence – since it does not actually matter)

`*  /`

unary `+  -`

higher

`**` except for a unary operator to the right of `**`

`2 * 10 + 5`

`*` has higher precedence than `+`

→ `2 * 10` is evaluated first

→ `20 + 5` → `25`

`**` has highest precedence in our previous list

`2 * 2 ** 3` → `2 * (2 ** 3)` → `2 * 8` → `16`

`-2 ** 4` → `-(2 ** 4)` → `-16`

(as opposed to `(-2) ** 4` → `16`)

→ except when unary operator is to the right of `**`

`2 ** -3` → `2 ** (-3)` → `0.125`

→ makes sense, difficult to interpret it otherwise anyway

A complete list of all operator precedence in Python can be found here:

https://docs.python.org/3/reference/expressions.html#operator-precedence

→ my advice

→ relying on operator precedence is tricky

→ very easy to introduce bugs

→ use parentheses

→ it's just a few keystrokes more and will save a lot of pain later!

use `(2 * 10) + 5` instead of `2 * 10 + 5`

Coding

Integer Division and Mod

Let's review long division!

$$
\begin{array}{r|l}
 & \;\;4\;3 \\
\hline
3 & 1\;3\;1 \\
 & 1\;2 \\
\cline{2-2}
 & \;\;1\;1 \\
 & \;\;\;\;\;9 \\
\cline{2-2}
 & \;\;\;\;\;2
\end{array}
$$

2 is the remainder

$131 \,/\, 3 \rightarrow 43\,\frac{2}{3}$

43 is the integer portion of the division

Python integer division: `//` `131 // 3` → `43`

Remainder: use Python mod operator `%` `131 % 3` → `2`

The // Operator

`a // b` calculates the "integer portion" of `a / b`

→ easy to understand when `a` and `b` are positive

Reality: `a // b` is the **floor** of `a / b`

`floor(x)` → the largest integer number `<= x`

`floor(-3.14)` → `-4` `floor(3.14)` → `3`

```
 12 / 5 → 2.4     12 // 5 → 2
-12 / 5 → -2.4   -12 // 5 → -3
```

The `mod` Operator

Again negative numbers complicates things a bit!

→ I said you can use `%` to calculate the remainder of dividing `a` by `b`

→ in this case, for **positive integers**, `a` and `b`

→ `a % b` and the remainder of dividing `a` by `b` is the same

→ intuitive for positive numbers

But `%` is defined for negative integers and even floats as well

→ what does that even mean?

The `mod` Operator

Let's go back to our first example $131\ /\ 3 \rightarrow 43\,\frac{2}{3}$

$$\frac{131}{3} = floor\left(\frac{131}{3}\right) + \frac{131\ mod\ 3}{3}$$

```
a / b = a // b + (a % b)/b
```

```
→ a % b = b (a / b - a // b)
```

```
→ a % b = a - b (a // b)
```

So `a % b` is **defined** as the value that satisfies the above equation

→ and that's how `a % b` is well-defined for negative values

→ and even for floats!

→ this explains the "weird" (aka non-intuitive) behavior for negative numbers

```
12 % 5 → 2    12 % -5 → -3   -12 % 5 → 3   -12 % -5 → -2
```

→ as well as how it works for real numbers `12.5 % 3 → 0.5`

```
12.5 // 3 → 4.0

a % b = a - b (a // b)

→ 12.5 % 3 = 12.5 - 3 (4.0) = 12.5 - 12.0 = 0.5
```

→ moral: be careful using "intuition" for `%` and `//` and negative values

→ fortunately most of the problems we work with involve positive integers

(more in coding video)

Coding

Comparison Operators

→ also know as relational operators

→ compares two things and yields a Boolean (`bool`) result

`==` equality comparison → `!=` for "not equal"

`<`, `<=`, `>`, `>=` assumes the operands are comparable

10.5 < 100 → makes sense

hello > 100 → doesn't really make sense

→ `==` between operands that are not comparable usually returns `False`

→ `<`, `<=`, etc between non-comparable operands usually generates an `Exception`

→ `TypeError` (we'll come back to exceptions later)

→ `int` and `float` types are comparable to each other

`10 <= 10.9` → `True`

Equality between integers is straightforward

`5 == 5` → `True`   `5 == 6` → `False`

`floats` are a different story!

`0.1 + 0.1 + 0.1 == 0.3` → `False`

→ in general: never use `==` to compare floats

What does it mean for two objects to be equal?

→ everything in Python is an object

`1` is an `int` object   `1.0` is a `float` object

are `1` and `1.0` the same object?   → No!

→ but they are the same value

→ need to differentiate what equality means

→ the object itself

→ the "value" (or state) of the object

Identity vs Value Equality of Objects

To see if two objects are the same **object** → `is`

To see if two (compatible) objects are equal in **value** (in some sense) → `==`

→ in most cases use `==`

→ we'll see situations where using `is` makes more sense

```
a = 1
b = 1.0
c = 1

d = 500
e = 500
```

`a == b` → `True`

`a is b` → `False`

`c is c` → `True`

`d == e` → `True`

but... `d is e` → `False` → `d` and `e` are not the same objects!

Identity vs Value Equality of Objects

The `is` operator is purely concerned with the memory address (identity) of the objects

→ `is` is called the identity comparison operator

The `==` operator, is, like `+`, actually implemented by the type itself

→ recall: `a + b` actually executes `a.__add__(b)`

→ `==` works the same way, using the `__eq__` method

`a == b` → `a.__eq__(b)`

So we can define what `==` means for custom types, by implementing `__eq__`

(we'll see this later in this course)

Other Comparison Operators

→ other comparison operators we'll cover in this course

→ membership operators: `in` and `not in`

→ works with collection types

→ determines membership in some collection

`s = {1, 2, 3.14, True, 5.1}` (like a mathematical set)

`1 in s` → `True`

`10 in s` → `False`

`10 not in s` → `True`

Coding

Boolean Operators

→ in Boolean algebra we only have two values: `True` and `False`

→ and three basic operators: `and`, `or`, `not`

→ Python syntax:

`not` is a unary operator

```
not True
not (a < b)
```

`and`, `or` are binary operators

```
True or False

True and False
```

```
(enabled == True) and (withdrawal <= balance)
```

The `not` Operator

→ `not` simply reverses the Boolean value

Truth Table

a	not a
True	False
False	True

The `and` Operator

→ `a and b` is `True` if and only if both `a` and `b` are `True`

→ `False` otherwise

a	b	a and b
True	True	True
True	False	False
False	True	False
False	False	False

notice something interesting:

→ if `a` is `False`, then `a and b` is always `False`, no matter what `b` is

The `or` Operator

→ `a or b` is `False` if and only if <u>both</u> `a` and `b` are `False`

→ `True` otherwise

a	b	a or b
True	True	True
True	False	True
False	True	True
False	False	False

notice something interesting:

→ if `a` is `True`, then `a or b` is always `True`, no matter what `b` is

Short-Circuited Evaluation

→ left and right operands are not restricted to values

→ can be expressions too e.g. `sin(a) > 0 and cos(a) < 0`

given a value `a`

calculate `sin(a)` → evaluate `sin(a) > 0`
→ result_1

calculate `cos(a)` → evaluate `cos(a) < 0`
→ result_2

evaluate `result_1 and result_2`

→ 4 calculations plus the `and` operation

→ but what if `result_1` had been False (i.e. `sin(a)` was not positive)?

→ recall: if `a` is `False`, then `a and b` is always `False`, no matter what `b` is

→ irrespective of what `cos(a) < 0` evaluates to, the result will always be `False`

→ so if the left operand evaluates to `False`, we don't even to calculate the right operand to get an answer → short-circuited evaluation

Short-Circuited Evaluation

The same happens with `a or b`

if `a` is True, then result is `True`, irrespective of what `b` is

→ Python returns `True` without evaluating `b`

And as we just saw with `a and b`

if `a` is False, then result is `False`, irrespective of what `b` is

→ Python returns `False` without evaluating `b`

→ short-circuited evaluation

→ can be very useful

→ will see examples of this in section on conditional execution

Example of Short-Circuiting Usefulness

→ suppose we have some trading algorithm that can calculate some buy signal (True/False)

→ the catch is that the calculation is complex and resource intensive

→ in addition, we only want to place an order if the exchange is open

we could write some code to do this:

```
if calc_signal(symbol) and exchange_open(symbol):
    buy(symbol)
```

→ problem: when exchange is closed we needlessly calculate the signal

→ but because of short-circuiting we can write:

```
if exchange_open(symbol) and calc_signal(symbol):
    buy(symbol)
```

→ this way if exchange is closed, we don't even calculate the signal

Coding

Conditional Execution

4

→ one of the fundamental constructs in programming is conditional execution

→ if something is true

→ run some code

→ else (optionally)

→ run some other piece of code

For example, for an ATM withdrawal:

→ if amount does not exceed available funds and does not exceed daily limit

  → dispense cash

  → print receipt

→ otherwise

  → deny request

  → display some text on screen

  → print slip containing reason

→ this is the primary reason we studied conditional expressions in the last chapter!

if… else…

The `if` Statement

```
if <expression evaluates to True>:
    code line 1
    code line 2
    …
```

note the colon!

notice how this code block is **indented**

→ this tells Python that all these lines should be executed if the condition is `True`

→ you "exit" a code block by unindenting your code

→ Python uses code indentation to group together chunks of code

→ called code blocks

→ if you are familiar with other languages such as Java or C/C++, this is equivalent to using braces `{}`

Examples

```
price = 200
if price < 250:
    make_purchase()
```

→ the call to `make_purchase()` will only be executed if `price < 250` evaluates to `True`, which in this case it is

```
price = 300
if price < 250:
    make_purchase()
```

→ in this case `make_purchase()` is not executed

Beware!

→ unindenting code from a block, "exits" the block

→ the following is a common mistake

```
price = 150
if price < 100:
    print('price is below 100, buying…')
make_purchase()
```

this is the code block

→ `price < 100` is `False`

→ does not run code in the `if` block

(`if` block only contains a **single** statement – the `print` statement)

→ runs `make_purchase()` → bug!!

The `else` Clause

→ often in conditional execution

if something is True

→ do something

otherwise

→ do something else

→ Python's `if` statement supports an `else` clause → it is optional

```
if <expression is True>:
    [Code Block 1]
else:
    [Code Block 2]
```

note how `else` is **unindented** from the `if` block, and followed by a colon

indent to form the `else` block

Example

```
price = 200

if price < 250:
    print('The price is right!')
else:
    print('Too pricey!')
print('Done.')
```

notice this line of code is unindented

→ it has nothing to do with the `if` or `else` blocks

→ it will always execute

→ `price < 250` is `True`

→ the `if` block is executed

→ the `else` block is skipped

→ code resumes after the `else` block

```
The price is right

Done
```

Example

```
price = 300

if price < 250:
    print('The price is right!')
else:
    print('Too pricey!')
print('Done.')
```

→ `price < 250` is `False`

→ the `if` block is skipped

→ the `else` block is executed

→ code resumes after the else block

```
Too pricey!
Done.
```

Nested `if` Statements

→ sometimes we need to nest conditional logic, either in the `if` block or in the `else` block

```
if price < 1000:
    if price < 500:
        volume = 50
    else:
        volume = 10
    make_purchase(volume)
else:
    print('Too pricey!')
```

→ if price < 1000 is True
- → if price < 500
  - → set volume to 50
- → otherwise
  - → set volume to 10
- → purchase specified volume

→ otherwise
- → Too pricey

Nested `if` Statements

→ the nesting can occur in the `else` block too

```
if price < 1000:
    make_purchase()
else:
    if price < 2000:
        contact_vendor()
    else:
        find_new_vendor()
```

→ can nest to any number of levels

→ too much nesting can make code hard to read!

→ keep it to a minimum

Coding

elif

Multi-Level `if` Statements

Consider this example to calculate a grade letter given a numeric grade:

```
if grade >= 90:
    grade_letter = 'A'
else:
    if grade >= 80:
        grade_letter = 'B'
    else:
        if grade >= 70:
            grade_letter = 'C'
        else:
            if grade >= 60:
                grade_letter = 'D'
            else:
                grade_letter = 'F'
```

→ that's a lot of nesting!

→ hard to read (for humans)

The `elif` Clause

Instead of this nested structure, Python provides an `elif` clause

→ equivalent to a nested `else-if`

→ does not require this double indentation

→ easier to read!

```
if grade >= 90:
    grade_letter = 'A'
elif grade >= 80:
    grade_letter = 'B'
else:
    grade_letter = 'F'
```

once an `if` or `elif` clause executes (is `True`)

→ **no other** `if`, `elif` or `else` block executes

`else` executes if no `if` or `elif` statement executed

Grade Letter Example

```
if grade >= 90:
    grade_letter = 'A'
else:
    if grade >= 80:
        grade_letter = 'B'
    else:
        if grade >= 70:
            grade_letter = 'C'
        else:
            if grade >= 60:
                grade_letter = 'D'
            else:
                grade_letter = 'F'
```

```
if grade >= 90:
    grade_letter = 'A'
elif grade >= 80:
    grade_letter = 'B'
elif grade >= 70:
    grade_letter = 'C'
elif grade >= 60:
    grade_letter = 'D'
else:
    grade_letter = 'F'
```

→ much more human readable!

Coding

Ternary Conditional Operator

Terminology

unary operator → an operator that takes a single operand

→ operator usually a prefix to the operand

→ -x

binary operator → an operator that takes two operands

→ usually operands are on either side of the operator

→ x + y

ternary operator → an operator that takes three operands

→ so how do we write that? 🤔

→ suppose we have an operator that takes three operands: a, b, c

→ the goal is for the operator to return `a + (b * c)`

→ this is a thing – it's called the Multiply-Accumulate operator (MAC)

(but not available in Python!)

→ maybe this? `a accmul b, c`

→ or maybe this? `a acc b mul c`

all we've done here is split the name of the operator into two
and added the operands in between

This type of conditional code is often used

```
if <conditional exp>:
    var = value1
else:
    var = value2
```

→ key is that each code block is a single assignment

→ to the same variable

→ Python introduces a conditional ternary operator to do this

The conditional ternary operator

→ remember that an operator operates on operands and returns (calculates) some result

```
if <conditional exp>:
    var = value1
else:
    var = value2
```

→ in this case we want the ternary operator's operands to be:

- → the conditional expression
- → the value to return if the expression is `True`
- → the value to return if the expression is `False`

The conditional ternary operator

```
if <conditional exp>:
    var = value1
else:
    var = value2
```

```
value1 if <conditional exp> else value2
```

→ this is a single ternary operator

→ if condition is `True`, it returns `value1`

→ if condition is `False`, it returns `value2`

Example

```
if price < 100:
    volume = 10
else:
    volume = 1
```

→ can be re-written using a conditional ternary operator

```
volume = 10 if price < 100 else 1
```

General Form

→ we saw examples where we used values as the return operands

→ but it's more general than that

→ the two value operands can be any expression

→ the result of the expression is then used

```
<exp1> if <condition> else <exp2>
```

```
var = (a - b) if a > b else (b - a)
```

Short-Circuiting

Just like we saw with Boolean operators, the ternary operator also uses short-circuit evaluation

```
<exp1> if <condition> else <exp2>
```

→ first evaluates <condition>

→ if it is True, evaluates and returns <exp1>

→ but does not evaluate <exp2>

→ if it is False, evaluates and returns <exp2>

→ but does not evaluate <exp1>

Example

```
result = a / b if b != 0 else 'NaN'
```

```
a = 10
b = 5
```

→ returns `2`

→ `b` is `5`, so `b != 0` evaluates to `True`

→ `a / b` is calculated and returned

```
a = 10
b = 0
```

→ this works just fine, and returns `NaN`

→ `b` is zero, so `b != 0` evaluates to `False`

→ `a / b` is **not** calculated

(thereby avoiding a division by zero exception)

→ `NaN` is returned

Coding

Sequence Types

5

What are Sequences?

→ sequences are ordered collections of objects

→ there is a first element

→ there is a second element

→ and a next one → sometimes called the sequential order

→ we can index those elements using integers

→ like counting them one by one

→ but in Python (and most other programming languages)

→ **numbering starts at `0`**

→ like anything in Python, **sequences are objects**

→ they just happen to be container type objects that contain other objects

Indexing Sequences

$\overbrace{\texttt{object, object, object, …, object}}^{n \text{ objects}}$ → length of sequence

```
object, object, object, …, object
  0       1       2        n-1
```

→ n objects in sequence

→ last element index is n-1

in this course:

→ first element refers to the element at index 0

→ second element refers to the element as index 1

→ last element refers to the element at index n-1 (assuming n elements in sequence)

Sequence Length

→ sequences are usually finite

→ but not all sequence types are

→ in this course we'll stick to finite sequences

→ first element

→ last element

→ finite length

Homogeneous vs Heterogeneous Sequences

certain sequence types can only contain objects that are all the same type

→ homogeneous sequence types

other types of sequences may contain objects that are of different type

→ heterogeneous sequence types

Sequence Types in this Chapter

`lists` → mutable heterogeneous sequence type

`tuples` → immutable heterogeneous sequence type

`strings` → immutable homogeneous sequence type

Lists

The `list` Type

→ it is a container type → it contains elements

→ it is a sequence type → elements are ordered sequentially
→ elements are indexed

→ lists can be heterogeneous

→ lists are mutable → can add, replace or remove elements

→ lists have unbounded growth → can add as many elements as we want
→ but they are still finite

→ lists are objects
- → they have state → the elements contained in the list
- → they have functionality → add element, remove element, etc

`list` Literals

→ Python lists can be created using literals

`[10, 20, 30, 40]`

note the enclosing square brackets `[]`

→ this is what indicates the type is a `list`

→ this list is homogeneous → all elements are integers

→ but they don't have to be `[10, 3.14, True]`

→ they can even be nested `[10, 20, 30, [True, False]]`

the last element of this list is itself a list

Accessing `list` Items by Index

→ lists are sequence types → sequential order → indexable

```
l = [10, 20, 30, 40, 50]
```
(length is 5)

index 0 1 2 3 4

→ we can reference an element by its index `l[i]`

`l[0]` → `10`

`l[1]` → `20`

`l[2]` → `30`

⚠ Trying to access a list by index greater than last index will cause an exception!

`l[5]` → `IndexError`

Sequence Length

`l = [10, 20, 30, 40, 50]`

→ visual inspection → length of `l` is `5`

→ but we can use code to calculate this for us

→ the `len` function

```
len(l) → 5

len([True, False]) → 2
```

Empty Lists

sometimes we want to start with an empty list

and have code that adds to the list as our program runs

→ to create an empty list we can just use a literal

```
l = []
```

then `len(l)` → `0`

⚠ `l[0]` → `IndexError`

Replacing a `list` Element

```
l = [10, 20, 30, 40, 50]
```

→ we can retrieve elements by index

`print(l[2])` → `30`

→ but we can also replace an element at index `i` with a different element

→ we use the assignment operator `=`

```
l[2] = True
l = [10, 20, True, 40, 50]
print(l[2])  → True
```

⚠ we are replacing elements – so the index must be valid!

`l[5] = 100` → `IndexError`

Coding

Tuples

The `tuple` Type

→ very similar to the `list` type

→ it is a container type

→ it is a sequence type

→ tuples can be heterogeneous

→ BUT… they are an immutable container type

→ unlike lists, once a tuple has been created

→ cannot add or remove elements

→ cannot replace elements

`tuple` Literals

→ Python tuples can be created using literals

```
(10, 20, 30, 40)
```

note the enclosing round brackets `( )`

→ this indicates the collection is a tuple

→ just like lists, they can can contain any object, including another tuple

```
(10, 20, (3, 4))

(10, 20, (True, False), [100, 200])
```

`tuple` Literals

→ often we don't even need the `( )`

→ Python interprets a comma separated list of elements as a tuple

→ so we can write `(10, 20, 30)`

→ or just `10, 20, 30`

→ both these code snippets result in `t` being a tuple

```
t = (10, 20, 30)

t = 10, 20, 30
```

`tuple` Literals

→ just like lists, tuples can contain any object

→ including other tuples or lists

$$\underbrace{\texttt{(1, }\underbrace{\texttt{[True, False]}}_{\text{list}}\texttt{, }\underbrace{\texttt{(3, 4)}}_{\text{tuple}}\texttt{)}}_{\text{tuple}}$$

→ we can omit the parentheses on the outer tuple

```
1, [True, False], (3, 4)
```

→ but not `(3, 4)`

```
1, [True, False], 3, 4
```

→ not the same

Indexing, Length

→ just like lists, elements can be read back from a tuple using an index number

→ the `len( )` function works with tuples also

```
t = 10, 20, 30, 40, 50

len(t) → 5

t[0] → 10

t[2] → 30

t[5] → IndexError
```

`tuples` are Immutable

→ unlike lists, we cannot replace an element of a tuple

```
t = 10, 20, 30
t[0] = 100
```
→ `TypeError`

→ the container is immutable

→ does not mean elements in the container are immutable

```
t = 10, 20, [True, True]
```
last element is a `list`

→ which is mutable

```
t[2] = 100
```
→ `TypeError`

```
t[2][1] = False
```
`t` → `10, 20, [True, False]`

Creating Empty `tuples`

→ not very useful, so not used very often

→ use empty parentheses

```
t = ()
```

→ that tuple is immutable, so it will remain empty for its lifetime

Coding

Strings

The `str` Type

→ this is also a container type

→ it is a sequence type

→ strings are homogeneous → they can only contain characters (unicode)

→ they are immutable

str Literals

→ Python strings can be created using literals

'this is a string'

note the enclosing quotes '…'

→ can also use double quotes

"this is a string"

note the enclosing double quotes "…"

→ these quotes/double-quotes are called the string delimiters

→ an empty string literal can be '' or ""

Indexing, Length

→ works the same way as any sequence type

→ use an index number to access elements of the string

→ use the `len( )` function to find the length of the string

```
s = 'Python'

len(s) → 6

s[0] → 'P'
```
(a string containing a single character)

```
s[1] → 'y'
s[5] → 'n'
s[6] → IndexError
```

Coding

Slicing

→ slicing is a way to extract ranges of elements from a sequence

→ start position (by index number)

→ stop position (by index number)

```
[start:stop]
```

→ start index is inclusive of the element

→ stop index is exclusive of the element

→ slices are the same type as the sequence being sliced

```
l = [10, 20, 30, 40]
      0   1   2   3
```

`l[0]` → `10` `l[1]` → `20` `l[2]` → `30` `l[3]` → `40`

`l[0:2]` → starts at `0`, and includes element at `0`

→ ends at `2`, but excludes element at `2`

`l[0:2]` → `[10, 20]` → result is also a (new) `list`

`l[1:3]` → `[20, 30]`

```
t = (10, 20, 30, 40)
      0   1   2   3
```

`t[0:2]` → `(10, 20)` → result is also a (new) `tuple`

`t[1:3]` → `(20, 30)`

→ `str` type is a sequence type → slicing for strings works the same way

```
s = 'Isaac Newton'
```

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

`s[0:4]` → `'Isaa'`

`s[0:5]` → `'Isaac'`

`s[6:9]` → `'New'`

Including Last Element in Slice

```
s = 'Isaac Newton'
     0 1 2 3 4 5 6 7 8 9 10 11
```

`s[6:11]` → `'Newto'`

→ how do we specify including the last element ?

→ it's ok to specify indexes outside the sequence bounds!

→ Python will automatically figure it out

`s[6:12]` → `'Newton'`

`s[6:1000]` → `'Newton'`

→ we can also leave the end index blank

→ Python will interpret as "up to and including the last element"

`s[6:]` → `'Newton'`

Including First Element in a Slice

```
s = 'Isaac Newton'
     0 1 2 3 4 5 6 7 8 9 10 11
```

→ just specify `0` as the start `s[0:5]` → `'Isaac'`

→ can also leave the start index blank

`s[:5]` → `'Isaac'`

→this is actually valid: `s[:]` → `'Isaac Newton'`

→ this made a shallow copy of the sequence

→ we'll come back to that in a bit

Slicing with Steps

→ a step is a way to specify an interval when slicing a sequence

```
s[start:stop:step]
```

`[2:10:2]` → start at (and include) index 2

→ end at (but exclude) index 10

→ move in steps of 2

2 3 4 5 6 7 8 9 → indexes: 2 4 6 8

```
l = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

0 1 2 3 4 5 6 7 8 9

`l[2:10:2]` → `[30 ,50 ,70 ,90]`

Negative Steps

→ possible to use negative step values

→ starts at index `start` (inclusive)

→ stops at index `end` (exclusive)

→ moves backwards → so `start` should be greater than `end`

```
l = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
      0   1   2   3   4   5   6   7   8   9
```

`l[9:6:-1]` → `[100, 90, 80]`

`l[:6:-1]` → `[100, 90, 80]`

`l[3::-1]` → `[40, 30, 20, 10]`

`l[::-1]` → `[100, 80, 80, 70, 60, 50, 40, 30, 20, 10]`

→ `strings` are sequence types

→ also works for strings

```
s = 'Isaac Newton'
     0 1 2 3 4 5 6 7 8 9 10 11
```

`s[11:5:-1]` → `'notweN'`

`s[:5:-1]` → `'notweN'`

`s[::-1]` → `'notweN caasI'`

`s[10::-2]` → `'owNcaI'`

`s[::-2]` → `'nte as'`

Coding

Manipulating Sequences

→mutable sequences can be modified

- → replace elements
- → delete elements
- → add elements
  - → often appended (to the end)
  - → can also specify where in the sequence to insert

Replacing Single Elements

Replace an element at index `i` by assigning a new element to that index

```
l = [10, 20, 30]

l[1] = 'hello'

l → [10, 'hello', 30]
```

Replacing an entire Slice

→ can also replace an entire slice

→ just assign a new collection to the slice

→ slice will be replaced with elements in RHS

```
my_list = [1, 2, 3, 4, 5]

my_list [0:3] = ['a', 'b']
my_list [0:3] = ('a', 'b')
my_list [0:3] = 'ab'
```

my_list → ['a', 'b', 4, 5]

→ Python uses the elements of the sequence in RHS when assigning to a slice

(but not when assigning using a single index)

Deleting Elements

→ can delete an element by index

```
my_list = [1, 2, 3, 4, 5]

del my_list[1]

my_list → [1, 3, 4, 5]
```

→ can delete an entire slice

```
my_list = [1, 2, 3, 4, 5]

del my_list[1:3]

my_list → [1, 4, 5]
```

Appending Elements

→ we can append one element

```
my_list = [1, 2, 3]
my_list.append(4)
```

```
    my_list → [1, 2, 3, 4]
```

→ to append multiple elements, we extend the sequence

```
my_list = [1, 2, 3]

my_list.extend(['a', 'b', 'c'])
my_list.extend(('a', 'b', 'c'))
my_list.extend('abc')
```

does the same thing

```
    my_list → [1, 2, 3, 'a', 'b', 'c']
```

Inserting an Element

→ instead of appending, we can insert at some index

→ use sparingly – this is much slower than appending or extending

```
my_list = [2, 3, 4, 5]
my_list.insert(0, 100)
my_list → [100, 2, 3, 4, 5]
```

```
my_list = [2, 3, 4, 5]
my_list.insert(2, 100)
my_list → [2, 3, 100, 4, 5]
```

→ element is inserted so its position is the index - remaining elements are shifted right

Coding

Copying Sequences

Shallow vs Deep Copies

→ two types of copies

→ shallow copies

→ new sequence is created (not same sequence object as original)

→ elements in new sequence reference the same elements as original

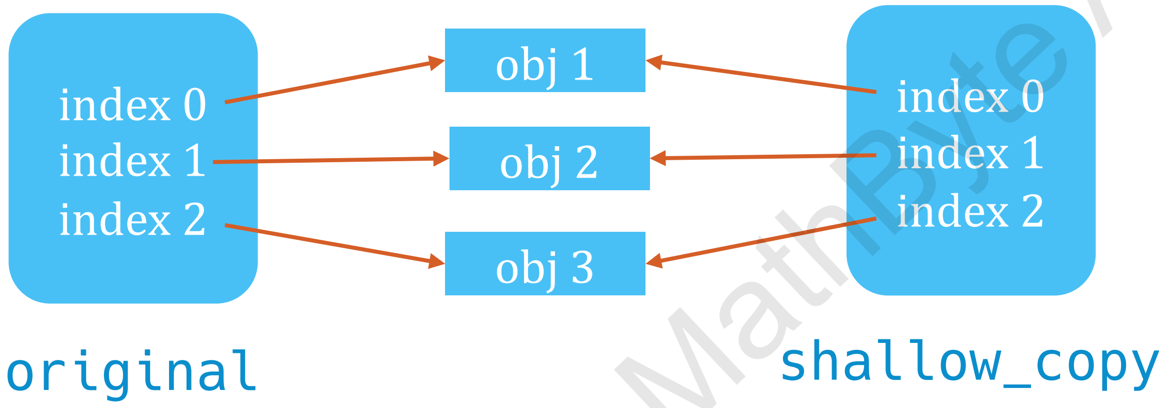
→ deep copies

→ new sequence is created (not same sequence object as original)

→ each element in new sequence is a deep copy of the original

→ totally new and independent objects

Shallow Copy

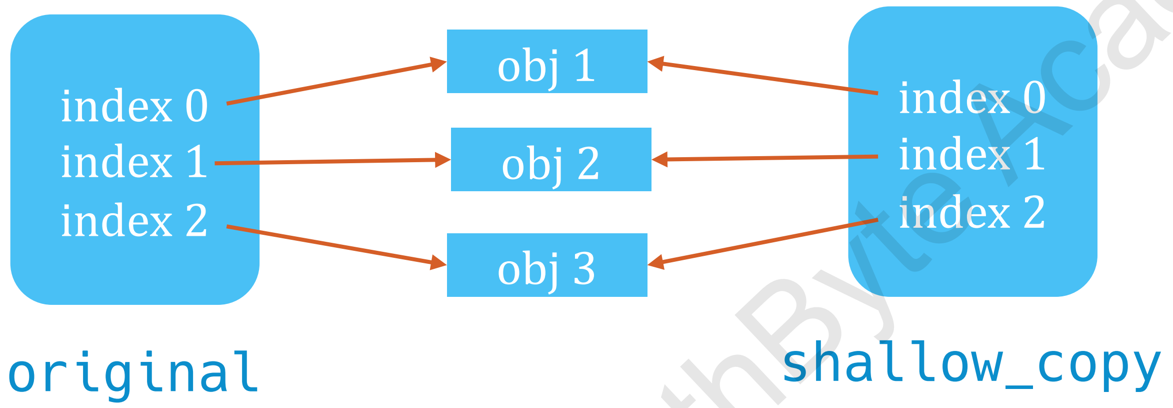


`original is shallow_copy` → `False`

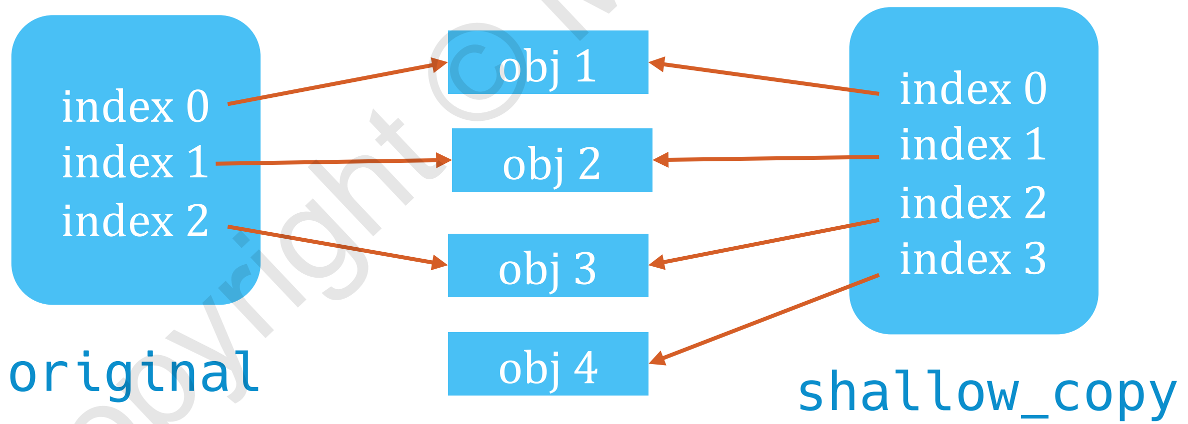
→ `original` and `shallow_copy` are **not** the same containers

→ but the elements are referencing the **same** objects

Shallow Copy

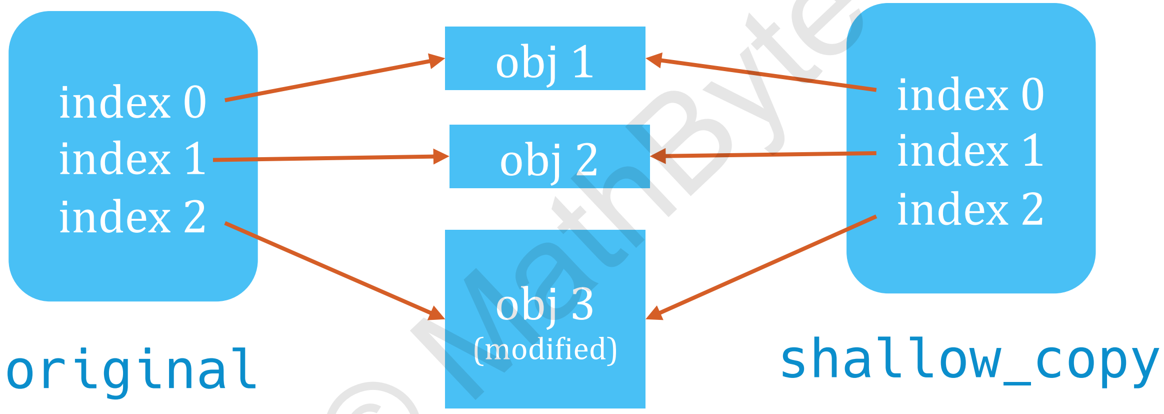


→ add/remove/replace element in one does not affect the other



Shallow Copy

→ but mutating an element will affect both (since it is a shared reference)



Creating Shallow Copies

→ use slicing to slice the entire sequence → `my_list[:]`

→ use the copy method → `my_list.copy()`

```
my_list = [1, 2, 3]

my_copy = my_list[:]
my_copy = my_list.copy()
```

my_copy → [1, 2, 3]

```
del my_copy[0]
```

my_copy → [2, 3]

my_list → [1, 2, 3]

Mutable Elements

```
my_list = [['a', 'b'], 2, 3]

my_copy = my_list.copy()
```

→ `my_list[0]` and `my_copy[0]` are both referencing the **same** list `['a', 'b']`

```
my_list[0] is my_copy[0] → True
```

so if we modify that element (from either sequence):

```
my_copy[0].append('c')

    my_copy[0] → ['a', 'b', 'c']
    my_list[0] → ['a', 'b', 'c']
```

Creating Deep Copies

→ uses `deepcopy` function in the `copy` module

```
from copy import deepcopy
my_list = [['a', 'b'], 2, 3]
my_copy = deepcopy(my_list)
```

`my_list[0] is my_copy[0]` → `False` → element has been copied too!

```
my_copy[0].append('c')
```

`my_copy[0]` → `['a', 'b', 'c']`

`my_list[0]` → `['a', 'b']`

Coding

Unpacking Sequences

consider a sequence

`data = (1, 2, 3)` → this is a `tuple` with three elements

we want to assign those values `1`, `2` and `3` to some symbols `a`, `b` and `c` resp.

→ could do it this way:

```
a = data[0]
b = data[1]
c = data[2]
```

but Python has a better way of doing this! unpacking

```
a, b, c = (1, 2, 3)
```

Since tuples don't actually need the parentheses in this case, we can write:

```
a, b, c = 1, 2, 3
```

→ this works with any sequence in general

```
a, b = [10, 20]       a → 10
                      b → 20

a, b, c = 'XYZ'       a → 'X'
                      b → 'Y'
                      c → 'Z'
```

→ beware!

→ number of elements in sequence on RHS must match number of symbols on LHS

```
a, b = 1, 2, 3
    → ValueError (too many values to unpack)

a, b, c = 1, 2
    → ValueError (not enough values to unpack)
```

Swapping Two Variable Values

→ this is a common problem
given two variables a and b, swap the value of a and b

Initial State: a → 10 End State: a → 20
b → 20 b → 10

→ typical solution uses a temporary variable

```
temp = a
a = b
b = temp
```

→ 3 lines of code and an unnecessary variable

Swapping Two Variable Values

→ can use unpacking to our advantage

→ remember: in an assignment, the RHS expression is evaluated completely first

  → then the assignment takes places

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

→ RHS is evaluated first `b, a` is the tuple `20, 10`

→ then the assignment is made `a, b = 20, 10`

→ values of `a` and `b` have been swapped!

Coding

Strings

6

→ strings are sequence types

→ but they are more specialized than generic sequences

→ they are homogeneous

→ each element is a single character

→ we have additional functionality available

Unicode

In the beginning…

… there was ASCII (American Standard Code for Information Interchange)

→ addressed the problem of a standard for assigning

→ numeric codes → to characters

→ printable and non-printable

→ and encoding the value into binary → using sequences of 7 bits

→ given a data stream filled with 0's and 1's

→ carve up in 7 bits and decode character

→ fonts handle displaying the character

→ a bunch of pixels

→ a glyph

→ supported character set was limited

→ 128 characters

→ 95 printable characters (a-z, A-Z, 0-9, * / etc)

→ 33 non-printable characters (control codes, e.g. esc, newline, tab, etc)

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x

→ attempts were made to extend the ASCII set

→ still far too limited

→ standard was poorly followed

→ Unicode was developed

→ focused on assigning a code to a character (code point)

→ does not specify how to encode the code points into a binary format

→ other standards for doing that appeared

→ UTF-8 ← very popular, default in Python

→ UTF-16

→ UTF-32 (utf → Unicode Transformation Format)

→ > 100,000 code points defined so far

Code Points

→ backward compatible with ASCII

ASCII character code for A → 65 (decimal), 41 (hexadecimal)

Unicode code point for A → 65 (decimal), 41 (hexadecimal)

decimal → base 10 (0 – 9)

hexadecimal → base 16 (0-9, A-F)

What is hex anyway?

Decimal system – uses powers of 10 → 10 digits, `0-9`

10^3	10^2	10^1	10^0
9	0	3	4

$$9034 = 4 \times 10^0 + 3 \times 10^1 + 0 \times 10^2 + 9 \times 10^3$$

Binary – uses powers of 2 → 2 digits, `0-1`

2^3	2^2	2^1	2^0
1	0	1	1

$$(1011)_2 = 1\times 2^0 + 1\times 2^1 + 0\times 2^2 + 1\times 2^3 = 11_{10}$$

Hexadecimal – uses powers of 16 → 16 digits, `0-9, A-F`

`A` → `10`, `B` → `11`, …, `F` → `15`

16^3	16^2	16^1	16^0
F	C	1	5

$$FC15 = 5\times 16^0 + 1\times 16^1 + 12\times 16^2 + 15\times 16^3$$
$$= 64533_{10}$$

Unicode Character A

https://www.compart.com/en/unicode/U+0041

Unicode Character “**A**” (U+0041)

the character (hex) code

A

Name:	Latin Capital Letter A [1]
Unicode Version:	1.1 (June 1993) [2]
Block:	Basic Latin, U+0000 - U+007F [3]
Plane:	Basic Multilingual Plane, U+0000 - U+FFFF [3]
Script:	Latin (Latn) [4]
Category:	Uppercase Letter (Lu) [1]
Bidirectional Class:	Left To Right (L) [1]
Combining Class:	Not Reordered (0) [1]
Character is Mirrored:	No [1]
GCGID:	LA020000 [5]
PostScript Name:	A [6]
HTML Entity:	`A` `A`
UTF-8 Encoding:	`0x41`
UTF-16 Encoding:	`0x0041`
UTF-32 Encoding:	`0x00000041`
Lowercase Character:	a (U+0061) [1]

the character name

corresponding lowercase letter

→ `ord( )` function

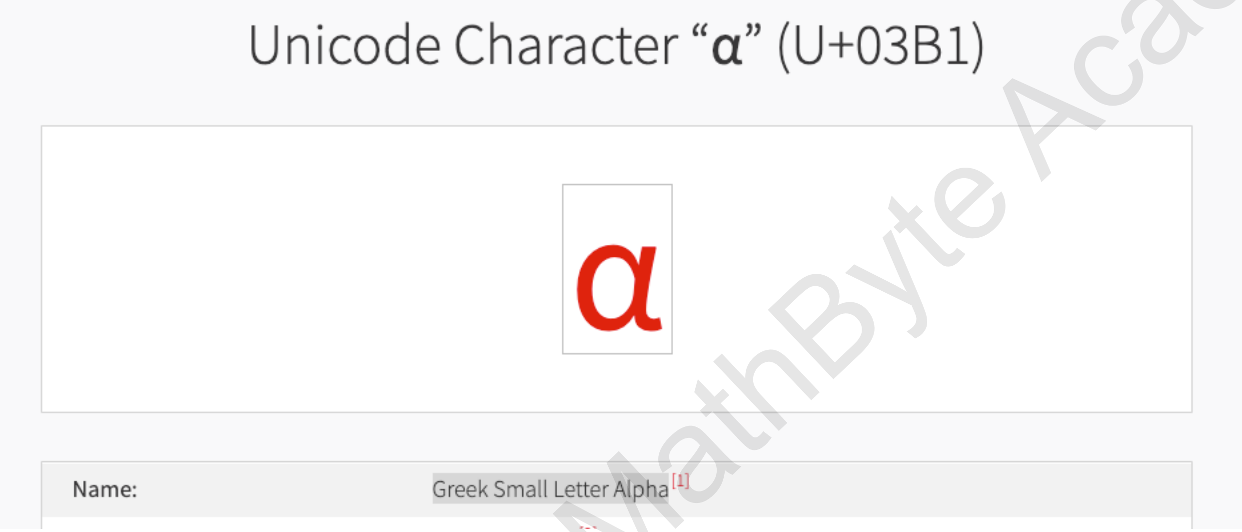
→ returns code point for a single character (in decimal)

`ord('A')` → `65`

→ `hex( )`

→ converts decimal to hex string

`hex(65)` → `'0x41'` (`0x` prefix indicates the number after that is in hex) https://www.compart.com/en/unicode/U+03B1




```
hex(ord("α")) → '0x3b1'
```

copy/paste the glyph from that page straight into your Python code

Other ways to specify the character in a string

→ use escape codes

→ by hex code → by name

`"\N{Greek Small Letter Alpha}"` → `"α"`

`"The letter \N{Greek Small Letter Alpha} is the first letter of the Greek alphabet."`

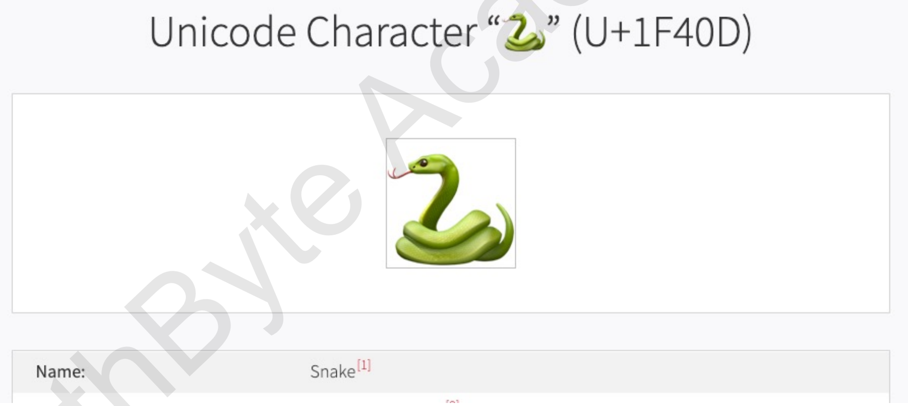
→ `'The letter α is the first letter of the Greek alphabet.'`

`"\u03b1"` `"\u03B1"` `\u` must be followed by exactly 4 hex digits (0-F)

`"The letter \u03B1 is the first letter of the Greek alphabet."`

→ `'The letter α is the first letter of the Greek alphabet.'`

https://www.compart.com/en/unicode/U+1F40D



`"\N{snake}"` → '🐍'

→note how character code 1F40D has 5 digits

→ must use `\U` followed by exactly 8 digits (`\u` is limited to 4 digits)

`"\U0001F40D"` → '🐍'

pad with zeroes to make 8 digits

Coding

Common String Methods

→ Python has a ton of string methods

https://docs.python.org/3/library/stdtypes.html#string-methods

In this video we are going to look at some in these categories

→ case conversions

→ stripping start and end characters

→ concatenating strings

→ splitting and joining strings

→ finding substrings

→ methods, called using dot notation `my_string.method()`

→ remember that strings are immutable

→ operations never modify a string → just return a new string

Case Mappings

```
lower()     'Hello World'.lower() → 'hello world'

upper()     'python'.upper() → 'PYTHON'

title()     'one two three'.title() → 'One Two Three'
```

→ returns a new string

→ primarily used for visual display

→ BEWARE: may not work for caseless comparisons

Case Folding

`casefold()` → used for caseless comparisons

```
s1 = 'hello'
s2 = 'HeLlo'
```

`s1.casefold() == s2.casefold()` → `True`

→ we'll explore this vs using case mappings in the code section

Stripping

sometimes we want to remove leading and trailing characters

→ trailing commas

→ whitespace around a string

`.lstrip()` → strips all whitespace on left of string

`.rstrip()` → strips all whitespace on right of string

`.strip()` → strips all whitespace on both ends of string

→ can specify what characters to strip

`.strip(' ')` → strip space characters from both ends

`.lstrip('abc')` → strip the characters `'a'`, `'b'`, `'c'` from left end

→ returns a new string

Concatenation

combining two or more strings to form a single string is called concatenation

```
'Hello' + '   ' + 'World!'  → 'Hello World!'
```

→ again, this creates a new string

Splitting Strings

→ useful for parsing data from a text file

```
data = '100, 200, 300, 400'
```

← a string containing comma delimited values

→ can easily split this on the comma

```
data.split(',')
```

→ returns a `list` of strings `['100', ' 200', ' 300', ' 400']`

note the spaces

→ we can `strip` them later

Joining Strings

→ this is the opposite of splitting strings

suppose we want to join these strings with `, ` characters between each:

```
'a'  'b'  'c'  'd'
```

we could write:

```
'a' + ', ' + 'b' + ', ' + 'c' + ', ' + 'd'
```

→ tedious to type out

→ "hardcoded"

→ what if we have a `sequence` of strings we want to concatenate

→ this approach is not general enough

Joining Strings

`data = ('ab', 'cd', 'ef')` → data is a sequence of strings

`'--'.join(data)` → join the strings in data with `--` in between
→ `'ab--cd--ef'`

`','.join(['10', '20', '30'])` → `'10,20,30'`

→ remember that a string is a sequence of single characters

`'='.join('python')` → `'p=y=t=h=o=n'`

Finding Substrings

→ often just want to know if a sequence of characters is contained inside another

→ use the `in` operator

```
'x' in 'xyz'         → True
'a' in 'xyz'         → False
'pyt' in 'python'    → True
'pyt' in 'Python'    → False
```

→ tests containment

→ but gives no indication of where the substring is

→ slight variation

→ does the string start (or end) with the specified characters

→ still a containment test

`.startswith('…')`   `.endswith('…')`

```
'python'.startswith('py') → True
'python'.startswith('hon') → False
'python'.endswith('py') → False
'python'.endswith('hon') → True
```

Finding the Index of a Substring

→ used when we need to know the index of the start position of a substring

```
data = 'This is a grammatically correct sentence.'
```

→ at what index does the string `'correct'` occur?

`data.index('correct')` → `24`

→ what if substring is not found?

→ Python raises a `ValueError`

→ potentially useful once we learn how to handle exceptions

Finding the Index of a Substring

→ what if we don't want an exception?

→ `find` → returns `-1` if substring is not found

```
data = 'This is a grammatically correct sentence.'

data.find('correct')      →  24

data.find('DOW')          →  -1
```

→ once we know how to handle exceptions, this method is a bit redundant

→ personally, I prefer using `index` and using exception handling

Important Note

→ only interested in whether or not a substring is contained in another string

→ use `in`

→ only use `index` or `find` when you need to know the index

→ `in` is much faster!

Coding

String Interpolation

→ often we want to build strings that contain values from some variable

→ can use concatenation

→ `+` works with two strings

→ cannot mix string and numeric for example

`'test' + 100 → TypeError`

`'test' + str(100) → 'test100'`

Example

Suppose we have four variables:

```
open_ = 98
high = 100
low = 95
close=99
```

We want to build a string that looks like this for display purposes:

```
'open: 98, high: 100, low: 95, close: 99'
```

→ using concatenation:

```
'open: ' + str(open_) + ', high: ' + str(high) + ', low: ' + str(low) + ', close:' + str(close)
```

→ tedious and error prone!

→ in fact there is an error, can you spot it?!

String Interpolation

→ multiple variants → two most common techniques

→ the `format` method

→ use `{}` as placeholders in our string

→ pass variables to `format` method in same order as we want them in the string

→ number of `{}` in string and arguments in `format` should match

→ `format` *can* have more arguments, they'll just be ignored

→ `IndexError` exception if not enough arguments

```
'open: {}, high: {}, low: {}, close: {}'.format(open_, high, low, close)
```

→ note how we did not have to convert the values to strings!

f Strings

→ new to Python 3.6

→ prefix the string with `f`

→ use `{expr}` directly inside the string

→ Python evaluates `expr` and interpolates the result directly inside the string

```
f'1 + 1 = {1 + 1}'  → '1 + 1 = 2'

value = 3.14
f'pi is approximately {value}'  → 'pi is approximately 3.14'

f'open: {open_}, high: {high}, low: {low}, close: {close}'

→'open: 98, high: 100, low: 95, close:99'
```

f Strings

→ could use do this as well

```
open_ = 98
high = 100
low = 95
close=99
```

```
f'open: {open_}, close: {close}, delta:{close - open_}'
```

→ `'open: 98, close: 99, delta: 1'`

Coding

Iteration

7

→ fundamental aspect of writing programs is repetition

→ want to repeat the same process (code) multiple times

→ how many times?

→ known in advance

→ load file with 10,000 rows

→ process each row → repeat the same process 10,000 times

→ deterministic

→ not known in advance

→ get commodity tick data

→ analyze data until ask price falls below some level

→ then do something else and stop processing

→ process may repeat 10 times, or 100 times, we don't know in advance

→ non deterministic

→ this repetition is called iteration

deterministic iteration

→ we iterate over the elements of some container

→ e.g. sequences

→ more generally over objects that are iterable

→ not all iterables are sequences

→a bag of marbles is iterable, but it is not a sequence!

→ `for` loop

non-deterministic iteration

→ we iterate while some condition is `True`

→ `while` loop

The `range` Function

The `range` Object

→ `range` object is an iterable object

→ it serves up integers one by one as they are requested

→ but the full list of integers does not exist all at once in memory

→ memory efficient

→ it has a finite number of integers

→ we can iterate over that range object

→ since it exists and has a finite number of integers → deterministic iteration

→ we can use the `range( )` function to create `range` objects

The `range()` Function

→ three flavors depending on how many arguments are specified

`range(end)` *(one argument)*

→ generates integers from `0` (inclusive) to `end` (exclusive)

`range(start, end)` *(two arguments)*

→ generates integers from `start` (inclusive) to `end` (exclusive)

`range(start, end, step)` *(three arguments)*

→ generates integers from `start` (inclusive) to `end` (exclusive)

→ in steps of `step`

→ should remind you of slicing

Viewing Contents of `range` Object

```
r = range(5)
```

`print(r)` → `'range(5)'`

→ not what we wanted

→ can convert `range` object to a `list` or `tuple`

`print(tuple(r))` → `(0, 1, 2, 3, 4)`

`print(list(r))` → `[0, 1, 2, 3, 4]`

Iteration

→ `range` object is iterable

→ we can use a `for` loop to iterate over the elements of this iterable (next lecture)

Coding

`for` Loops

→ `for` loops are used to iterate over elements of any iterable

- → the loop mechanism retrieves elements from the iterable one at a time
- → the body of the for loop is executed for each element retrieved
- → the loop terminates when all elements have been iterated

```
for x in ['a', 'b']:
    y = x + x
    print(y)
print('done')
```

note how the body is indented

→ just like `if…else…` code blocks

unindented → not in loop body

1st iteration:

`'a'` is retrieved and assigned to the symbol `x`

`y` is the concatenation of `x` and `x` → `'aa'`

`'aa'` is printed to the console

2nd iteration:

`'b'` is retrieved and assigned to the symbol `x`

`y` is the concatenation of `x` and `x` → `'bb'`

`'bb'` is printed to the console

3rd iteration: → no more elements → loop terminates

→ code after loop executes

→ `'done'`

Iterating over `range` Objects

→ `range` objects are iterable

```
for i in range(4):
    sq = i * i
    print(sq)
```

`range(4)` → `0, 1, 2, 3`

output:

```
0
1
4
9
```

Loop Bodies (Blocks)

→ block can contain any valid Python code

→ `if…else…`

→ another loop (nested loop)

```
for i in range(1, 4):
    for j in range(1, i+1):
        print(i, j, i*j)
    print('')
```

```
1 1 1

2 1 2
2 2 4

3 1 3
3 2 6
3 3 9
```

```
i = 1
j in range(1, 1+1)

i = 2
j in range(1, 2+1)

i = 3
j in range(1, 3+1)
```

```
data = [10, 20, 30, -10, 40, -5]
```

suppose we want to replace any negative value with `0`

→ we can iterate over the data and test for negative numbers:

```
for number in data:
    if number < 0:
        number = 0
    print(number)
```

```
10
20
30
0
40
0
```

→ but how do we replace `-10` and `-5` with `0` ?

→ easy if we know the index number

```
data[3] = 0
data[5] = 0
```

→ but we don't know that!!

The **enumerate** Function

`enumerate` is a function that

→ takes an iterable argument

→ returns a new iterable whose elements are a tuple consisting of:

→ the index number of the original element

→ the original element itself

```
data = [10, 20, 30, -10, 40, -5]
for t in enumerate(data):
    print(t)
```

```
(0, 10)
(1, 20)
(2, 30)
(3, -10)
(4, 40)
(5, -5)
```

→ at each iteration `t` is a tuple `(index, element)`

→ it can be unpacked!

```
data = [10, 20, 30, -10, 40, -5]
for t in enumerate(data):
    index, element = t
    if element < 0:
        data[index] = 0
```

```
data → [10, 20, 30, 0, 40, 0]
```

→ but we can do one better → we can unpack in the `for` clause itself

```
for index, element in enumerate(data):
    if element < 0:
        data[index] = 0
```

Coding

`while` Loops

→ different than `for`

→ here we want to repeat some code as long as some condition is `True`

→ non-deterministic → we don't necessarily know when condition becomes `True`

→ maybe never! → infinite loop

```
while expr:
    <code block>
```

→ `expr` is evaluated at the start of each iteration

→ if it is `True`, execute `<code block>`

→ if it is `False`, terminate loop immediately

→ may never execute (if `expr` is `False` on first iteration)

→ may never terminate (if `expr` never becomes `False`)

```python
value = 10

while value < 15:
    print(value)
    value = value + 1
```

increments `value` by `1`

output:

```
10
11
12
13
14
```

```python
value = 100

while value < 15:
    print(value)
    value = value + 1
```

output: `no output`

```
value = 10

while value < 15:
    print(value)
    value = value - 1
```

decrements `value` by `1`

output:

```
10
9
8
7
6
…
```

infinite loop!!

Coding

`continue`, `break`, `else`

Skipping an Iteration

→ sometimes we want to skip an iteration, but without terminating the loop

→ `continue`

→ immediately jumps to the next iteration

```
my_list = [1, 2, 3, 100, 4, 5]

for i in my_list:
    if i > 50:
        continue
    print(i)
print('done')
```

```
1
2
3
4
5
'done'
```

→ when `i` is `100`

→ `continue` is executed

→ loop jumps to next iteration

→ `continue` is not used too often

→ can sometimes make code difficult to read/understand

```
for i in my_list:
    if i > 50:
        continue
    print(i)
print('done')
```

→ equivalently:

```
for i in my_list:
    if i <= 50:
        print(i)
print('done')
```

→ less code, easier to read/understand

Early Termination

→ loops can be exited early (before all elements have been iterated)

→ `break`

```
my_list = [1, 2, 3, 100, 4, 5]

for i in my_list:
    if i > 50:
        break
    print(i)
print('done')
```

```
1
2
3
'done'
```

→ when `i` is `100`

→ `break` is executed

→ loop is terminated immediately

Early Termination

→ loop terminating early using `break`

→ sometimes called abnormal or early termination

→ sometimes want to execute some code if loop terminated normally

→ and different code otherwise (early/abnormal termination)

Example

We are scanning through an iterable, looking for an element equal to 'Python'

If we find the value, we want to terminate our scan immediately, and print `'found'`, otherwise we want to print `'not found'`

```
found = False

for el in my_list:
    if el == 'Python':
        found = True
        print('found')
        break

if not found:
    print('not found')
```

The `else` Clause

→ Python is really confusing here…

`for` loops can have an `else` clause

→ but it has nothing to do with the `else` clause of an `if` statement

→ the `else` clause of a `for` loop executes if and only if no `break` was encountered

in my mind I read it as "else if no break"

```
for i in range(5):
    <code block 1>
else: # if no break
    <code block 2>
```

→ `<code block 2>` executes if loop terminated normally
(i.e. no `break` encountered)

Back to our Example

```
found = False

for el in my_list:
    if el == 'Python':
        found = True
        print('found')
        break

if not found:
    print('not found')
```

equivalently:

```
for el in my_list:
    if el == 'Python':
        print('found')
        break
else:  # if no break
    print('not found')
```

Coding

Dictionaries

8

Dictionaries are one of the most important data structures in Python

→ we don't always see them

→ but they're lurking in the shadows! 😎

→ we saw that variables are symbols pointing to objects

→ some string (variable name) is associated with some object

objects are also dictionaries

→ properties are symbols associated to some value (object)

→ methods are names associated to some function

`s.upper()`   `l.append()`

→ associating two things together is extremely useful

a phone book → associates a number to a name

DNS → associates a URL with a numeric IP address

book index → associates a chunk of text with a page number

→ associative arrays → sometimes called a map

→ abstract concept

→ can be implemented in different ways

→ Dictionaries (or hash maps) are one concrete implementation

Associative Arrays and Dictionaries

Associating Things

→ ASCII table → associates a numeric value to certain characters

```
A → 65      a → 97       space → 32
B → 66      b → 98       < → 60
…           …            @ → 64
Z → 90      z → 122      …
```

We could try this:

```
keys = [' ', '<', '@', 'A', 'B', …, 'Z', 'a', 'b', …, 'z']
values = [32, 60, 64, 65, 66, …, 90, 97, 98, …, 122]
```

→ to find the numerical value of `'A'`

→ scan `keys` to find index of `'A'`

→ lookup the value for that index in the `values` list (array)

Another Approach

instead of storing the data in separate lists, use a list of tuples

→ each tuple has two elements → `(key, value)`

```
items = [('A', 65), … , ('Z', 90), ('a', 97), … , ('z', 122)]
```

→ to find value associated with `'a'`

→ scan items looking at first item of tuple until we find `'a'`

→ the value we want is the second element of that tuple we just found

→ both approaches have one major drawback

→ must scan an array until we find the correct element

→ the longer the array, the longer time this will take (worst case is last element)

Hash Maps (aka Dictionaries)

→ better implementation is the hash map (or dictionary)

→ similar to the last approach we saw

→ but a special mechanism is used to quickly find a key

→ lookup speed is not affected by size of dictionary

IMPORTANT → keys must be hashable (hence the term hash map)

→ what that means exactly is not important now

→ strings are hashable → numerics are hashable

→ tuples *may* be hashable (if all the elements are themselves hashable)

→ lists are not hashable (in general, mutable objects are not hashable)

Python Dictionaries

→ a dictionary is a data structure that associates a value to a key

→ both value and key are Python objects

→ key must be hashable type (e.g. `str`, `int`, `bool`, `float`, ...) and unique

→ value can be any type

→ type is `dict`

→ it is a collection of key: value pairs

→ it is iterable

→ but it is not a sequence type

→ values are looked up by key, not by index

→ technically there is no ordering in a dictionary

(we'll come back to this point!)

→ it is a mutable collection

Dictionary Literals

→ dictionaries can be created using literals

```
d = {'a': 97, 'b': 98, 'A': 65, 'B': 66, 'z': 122, 'Z': 90}
```

→ we can use a single line, but often we structure it over multiple lines to make it more readable

→ readability matters!

```
d = {
    'a': 97,
    'b': 98,
    'A': 65,
    'B': 66,
    'z': 122,
    'Z': 90
  }
```

Looking up values in a Dictionary

→ use `[]` just like for sequence types

→ but instead of an index value we specify the key

```
d = {
  'a': 97,
  'b': 98,
  'A': 65,
  'B': 66,
  'z': 122,
  'Z': 90
 }
```

`d['a']` → `97`

`d['Z']` → `90`

Replacing the Value of an existing Key

```
d = {
  'symbol': 'AAPL',
  'date': '2020-03-10',
  'close': 285
}
```

To change the value associated to the key `'close'`:

```
d['close'] = 285.34
```

→ dictionary now looks like this:

```
d = {
  'symbol': 'AAPL',
  'date': '2020-03-10',
  'close': 285.34
}
```

Adding a New key:value Pair

→ simply assign a value to a new key

→ if key exists, it will be updated as we just saw

→ if key does not exist, a new entry is inserted with key and value

(and this explains why keys in a dictionary are necessarily unique!)

```
d = {
  'symbol': 'AAPL',
  'date': '2020-03-10',
  'close': 285.34
}
```

```
d['open'] = 277.14
```

→

```
d = {
  'symbol': 'AAPL',
  'date': '2020-03-10',
  'close': 285.34,
  'open': 277.14
}
```

Deleting a key:value Pair

→ we can remove key:value pairs from a dict

→ use the `del` keyword

```
d = {
  'symbol': 'AAPL',
  'date': '2020-03-10',
  'close': 285.34,
  'open': 277.14
}
```

```
del d['open']
```

→

```
d = {
  'symbol': 'AAPL',
  'date': '2020-03-10',
  'close': 285.34}
```

Common Exceptions

certain operations on dictionaries can lead to `KeyError` exceptions

→ trying to read a non-existent key

→ trying to delete a non-existent key

trying to use a non-hashable object as a key leads to a `TypeError` exception

```
d[[10, 20]] = 100
```

→ `TypeError: unhashable type: 'list'`

→ `[10, 20]` is a `list`, and lists are not hashable

→ cannot be used as a key

Coding

Iterating Dictionaries

Dictionaries are Iterable

→ means we can use a `for` loop to iterate over… what?

keys? values? key:value pairs?

→ turns out, any of the above

→ default iteration is over the dictionary keys

```
data = {'a': 1, 'b': 2, 'c': 3}
```

```
for k in data:
    print(k)
```

→

```
a
b
c
```

Iterating over values

→ dictionaries have a method called `values()`

→ `values()` returns an iterable containing just the values of the dictionary

```
data = {'a': 1, 'b': 2, 'c': 3}

for v in data.values():
    print(v)
```

→

```
1
2
3
```

Iterating over key:value Pairs

→ dictionaries have a method called `items()`

→ `items()` returns an iterable containing the keys and values in a tuple

```
data = {'a': 1, 'b': 2, 'c': 3}

for t in data.items():
    print(t)
```

→

```
('a', 1)
('b', 2)
('c', 3)
```

→ remember unpacking?

```
for k, v in data.items()
    print(f'{k} = {v}')
```

→

```
a = 1
b = 2
c = 3
```

The `keys()` Method

Technically there is also a `keys()` method

→ behaves like `values()` or `items()`

→ but it is an iterable over the keys of the dictionary

```
data = {'a': 1, 'b': 2, 'c': 3}

for k in data.keys():
    print(k)
```

→

```
a
b
c
```

→ but default iteration is over the keys anyway

→ so `keys()` is not particularly useful for iteration

Insertion Order

→ we saw in sequence types that elements have positional order

→ not every iterable has positional order

→ we can pull marbles out of a bag, but there is no particular order

→ for a long time Python dictionaries were the same

→ a "bag" of key:value pairs that could be looked up by key

→ iteration order was not guaranteed to be anything specific

→ changed in Python 3.6

→ the iteration order reflects the insertion order

Insertion Order

what does insertion order mean?

```
d = {'z': 100, 'a': 1, 'b': 2}
```

→ literal: insertion order is the order in which the key:value pairs are listed out

→ `'z': 100, 'a': 1, 'b': 2`

→ adding a new element `d['x'] = 98`

→ `'x': 98` was added last, so right now it's the "last" element

→ `'z': 100, 'a': 1, 'b': 2, 'x': 98`

→ but we still cannot retrieve elements by index

Coding

Working With Dictionaries

Membership Testing

→ can test if a key exists in a dictionary using `in`

```
d = {'a': 1, 'b': 2}
```

`'a' in d` → `True`

`'x' in d` → `False`

→ `not in` can be used to test if a key is **not** present

Useful Methods and Functions

`d.clear()` → removes all elements from `d`

`d.copy()` → creates a shallow copy of `d`

- → same as sequences
- → use `copy.deepcopy()` to create a deep copy

`len(d)` → returns number of elements in `d`

Other Methods to Create Dictionaries

```
d = {'a': 1, 'b': 2}

d = dict(a = 1, b = 2)
```

→ symbols must be valid variable names and will be used, in string form, as the keys

→ can create a dictionary with several keys all initialized to the same value

```
d = dict.fromkeys(['cnt_1', 'cnt_2', 'cnt_3'], 0)

    d → {'cnt_1': 0, 'cnt_2': 0, 'cnt_3': 0}
```

→ first argument of `fromkeys( )` should be an iterable (list, tuple, string, etc)

```
d = dict.fromkeys('abc', 100)
    d → {'a': 100, 'b': 100, 'c': 100}
```

Creating Empty Dictionaries

→ often we create dictionaries that start empty

→ and get mutated (modified) as our code runs

→ can use a literal `d = {}`

→ can use the `dict()` function `d = dict()`

The `get( )` Method

→ trying to retrieve a non-existent key results in a `KeyError` exception

→ sometimes we want to have a "default" value if a key does not exist

→ could use if statements and test is key exists using `in`

→ could try to retrieve the key and handle the exception

→ or, use the `get( )` method

`get( )` can take two arguments

→ the key for which we want the corresponding value

→ the default value we want to use if key does not exist

→ `get( )` can take a single argument, the key

→ default value is `None` (special object to indicate "nothing")

The `get()` Method

```
d = {'length': 10, 'width': 20}
```

`d.get('length', 0)` → `10`

the key exists, so the corresponding value (`10`) is returned

`d.get('height', 0)` → `0`

the key does not exist, so the default (`0`) is returned

`d.get('height')` → `None`

the key does not exist, so the default default-value (`None`) is returned

The `get()` Method

→ data in Python is often handled using dictionaries

when we work with data we often have missing values

sometimes, not only is the value missing, but the key as well

→ using `get()` allows us to simplify our code to assign a default for missing keys

```
if 'ssn' in person_dict:
    social = person_dict['ssn']
else:
    social = ''
```

→ `social = person_dict.get('ssn', '')`

Merging one Dictionary into Another

→ the `update( )` method

→ takes a single argument: another dictionary

`d1.update(d2)`

the `key:value` pairs of `d2` will be merged into `d1`

→ keys in `d2` not in `d1` will be added to `d1` (with the value)

→ keys in `d2` that are present in `d1` will overwrite the value in `d1` with that of `d2`

→ Important: `d1` is mutated

Merging one Dictionary into Another

```
d1 = {'a': 1, 'b': 2}

d2 = {'b': 30, 'c': 40}
```

```
d1.update(d2)    d1 → {'a': 1, 'b': 30, 'c': 40}

d2.update(d1)    d2 → {'b': 2, 'c': 40, 'a': 1}
```

Coding

Sets

9

What are Python sets?

→ just like a mathematical set

→ a collection of elements

→ no ordering to the elements

→ each element is unique

→ it is an iterable

→ but no guarantee on what the iteration order will be

think of it like a bag of marbles (a collection of marbles)

to iterate you reach in the bag and grab a marble (any marble)

continue doing so until the bag is empty

→ no order guaranteed! → each marble is unique!

Just like mathematical sets, Python supports set operations

→ union

→ intersection

→ difference

→ membership (is some object an element of a set or not)

→ containment (subset, strict subset, superset, strict superset)

→ if you're a little rusty on sets, you should brush up before proceeding with this section

Python Sets

think back to keys in a dictionary

→ they are unique

→ they are iterable

→ they have no particular order (well, Python 3.6 maintains insertion order)

→ keys can be added or removed (dictionary is mutable)

→ they are hashable too - but leave that aside for a moment

does that remind you of a set?

→ we can think of the keys in a dictionary as a set

→ Python's implementation of sets is essentially like a dictionary

→ but no values, only keys

→ because of this, set elements must be hashable too

Python Sets

→ type is `set`

→ sets are iterable

→ iteration order is not guaranteed (at least not yet)

→ set elements must be hashable

→ sets are mutable

  → sets are not hashable

    → a set cannot be an element of another set, or a key in a dictionary

    → if you really want nested sets, use `frozenset`

      → immutable equivalent of sets – those are hashable
      (if all the elements are, themselves, hashable)

Defining Sets

→ literal form `{1, 'a', True}`

→ note the `{}` – just like for dictionaries

→ but no `key:value` pairs, just the "`keys`"

→ can also use the `set( )` function

`set([1, 'a', True])`

→ empty set

→ cannot use `{}`

→ that would be an empty dictionary

→ `set( )`

Defining Sets

→ can also make a set from any iterable (of hashable elements)

```
l = [1, 2, 3, 4, 5]
s = set(l)   s → {1, 2, 3, 4, 5}

l = [1, 1, 2, 2, 3, 3, 4, 4, 5, 5]
s = set(l)   s → {1, 2, 3, 4, 5}


s = set('python')  s → {'p', 'y', 't', 'h', 'o', 'n'}

s = set('parrot')  s → {'p', 'a', 'r', 'o', 't'}
```

→ use a `for` loop for iteration

→ use `in` for membership testing

→ `len(s)` returns the number of elements in the set

→ `s.clear()` removes all the elements of the set

→ `s.copy()` creates a shallow copy

Coding

Common Set Operations

Disjointedness

→ two sets are disjoint if they have no elements in common

`s1.isdisjoint(s2)`

→ `True` if no common elements exist

→ `False` if one or more common elements exist

(two elements `a` and `b` are considered the same if `a == b` is `True`)

Adding and Removing Elements

```
s = {10, 'b', True}
s.add(4)         s → {10, 'b', True, 4}
s.add('b')       s → {10, 'b', True, 4}   → no duplicates
                                            in a set
s.remove('b')    s → {10, True, 4}
s.remove(100)     → KeyError

s.discard(4)     s → {10, True}
s.discard(100)    → no exception   s → {10, True}
```

Subsets and Supersets

`s1 < s2` → `True` if `s1` is a strict subset of `s2`

`s1 <= s2` → `True` if `s1` is a subset of `s2`

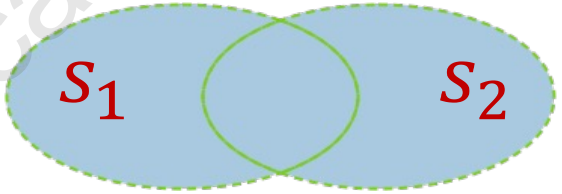
`s1 > s2` → `True` if `s1` is a strict superset of `s2`

`s1 >= s2` → `True` if `s1` is a superset of `s2`

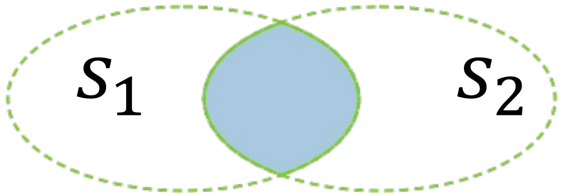
```
{1, 2} < {1, 2, 3} → True        {1, 2} < {1, 2} → False
{1, 2} <= {1, 2, 3} → True       {1, 2} <= {1, 2} → True

{1, 2, 3} > {1, 2} → True        {1, 2} > {1, 2} → False
{1, 2, 3} >= {1, 2} → True       {1, 2} >= {1, 2} → True
```

Unions and Intersections

`s1 | s2` → returns the union of `s1` and `s2`



`s1 & s2` → returns the intersection of `s1` and `s2`



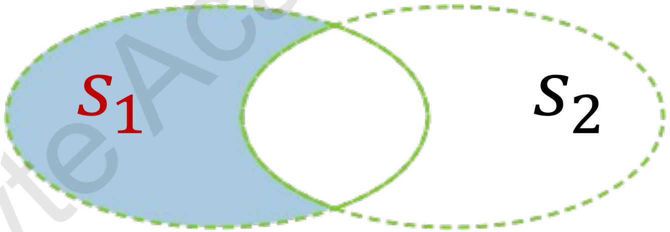

```
s1 = {1, 2, 3}
s2 = {3, 4, 5}
```

`s1 | s2` → `{1, 2, 3, 4, 5}`

`s1 & s2` → `{3}` (again, set elements are unique)

Set Difference

the difference `s1 - s2` of two sets is all the elements of one set minus the elements of the other set



→ caution: set difference is not commutative `s1 - s2 ≠ s2 - s1`

```
s1 = {1, 2, 3}
s2 = {3, 4, 5}
```

`s1 - s2` → `{1, 2}`

`s2 - s1` → `{4, 5}`

→ always keep sets in mind when coding

→ membership testing with sets is much faster than lists or tuples

→ easy to eliminate duplicate values from a collection

→ easy to find common values between two collections

→ easy to find values in one collection but not in another

Coding

Comprehensions

10

→ comprehensions are an easy way to create new iterables from other iterables

→ like using a loop, but easier and more concise syntax

→ works well for simple cases

→ can quickly become unreadable!

→ readability matters!!

Example

given a list of 2D vectors

```
[(0, 0), (1, 1), (1, 2), (3, 5)]
```

create a new list containing the magnitude of each vector

→ $[0^2 + 0^2, 1^2 + 1^2, 1^2 + 2^2, 3^2 + 5^2]$

→ [0, 2, 5, 34]

List Comprehensions

- a comprehension is a way to use one iterable to create another
  - more concise than using regular for loops
    - use for simple computations
    - comprehensions can quickly become confusing

- different types of comprehensions
  - lists
  - dictionaries
  - sets
  - generators

List Comprehensions

a list comprehensions is used to generate a `list` object

Example

we start with an iterable of numbers:

```
num = (1, 2, 3, 4, 5)   or   num = [1, 2, 3, 4, 5]
```

want to create a new list containing the square of each element

```
sq = [1, 4, 9, 16, 25]
```

→ can do this without comprehensions

```
numbers = (1, 2, 3, 4, 5)
sq = []
for number in numbers:
    sq.append(number ** 2)
```

this is the new list we want to create

loop through every element of the `numbers` iterable

calculate the square of the number and append it to the new list

```
sq → [1, 4, 9, 16, 25)
```

→ or we can use a comprehension

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

[ ] indicates we are creating a `list`

```
sq = [number ** 2 for number in numbers]
```

an expression used to calculate each element of the new list

iteration over existing iterable - note how the loop variable is available in the expression to the right

→ in general `[expression for item in iterable]`

→ comprehensions offer a more concise (and more efficient!) way of creating one iterable from another

→ in terms of result, these two things do the same

```
sq = []
for number in numbers:
    sq.append(number ** 2)
```

```
sq = [number ** 2 for number in numbers]
```

→ comprehensions are actually functions

→ builds up and returns the calculated iterable what about something like this?

given an iterable of integers

→ generate a new list that only contains the even integers

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
```

→ generate `evens = [2, 4, 6, 8]`

→ can use a "standard" approach

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
evens = []
for number in numbers:
    if number % 2 == 0:
        evens.append(number)
```

→ comprehension syntax supports an `if` clause

```
evens = [number for number in numbers if number % 2 == 0]
```

→ in general

```
[expression1 for item in items if expression2]
```

optional – acts like a filter

Coding

Dictionary/Set Comprehensions

→ similar to list comprehensions

→ use `{}` instead of `[]`

→ remember literals for dictionaries and sets use `{}`

→ dictionary elements are pairs → `key:value`

→ set elements are single values

```
d = {'a': 1, 'b': 2, 'c': 3}

s = {'a', 'b', 'c'}
```

Dictionary Comprehension

```
{key: value for item in items if expr}
```

can be any valid Python expression that
calculates some value

can be any Python expressions that
calculates a valid key

Example

Given two lists one of which contains widget names, the other containing the sales number for each of those widgets – ordered the same

```
widgets = ['widget 1', 'widget 2', 'widget 3', 'widget 4']
sales = [10, 5, 15, 0]
```

→ create a dictionary whose keys are the widget names, and the value the number of sales, but only include widgets that had sales.

→ "traditional" approach

```
d = {}
for i in range(len(widgets)):
    if sales[i] > 0:
        d[widgets[i]] = sales[i]
```

Example

→ or, we can use a comprehension instead

```
widgets = ['widget 1', 'widget 2', 'widget 3', 'widget 4']
sales = [10, 5, 15, 0]
d = {
      widgets[i]: sales[i]
      for i in range(len(widgets)
      if sales[i] > 0)
}
```

→ later we'll see an even easier way to do this

→ compare to "traditional" approach

```
d = {}
for i in range(len(widgets)):
    if sales[i] > 0:
        d[widgets[i]] = sales[i]
```

Set Comprehensions

→ similar to a dictionary comprehension

→ but elements are not `key: value` pairs

→ just the "key" portion

```
{expr1 for item in items if expr2}
```

can be any valid Python expression that calculates some value

Example

Given a list of integers, create a set that contains a unique collection of the squares of just the even integers

```
numbers = [1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6]


s = {number ** 2
     for number in numbers
     if number % 2 == 0
}
```

Coding

Exceptions

11

What are exceptions?

→ exceptions are special events that happen when something out of the ordinary happens while our code is running

→ an exception is generally unexpected behavior

  → but not always

  → it may be something we expect to happen from time to time

    → we can deal with it and continue running our code

→ so an exception is not necessarily an error

→ but unhandled exceptions will cause our program to terminate

Terminology

exception → a special type of object in Python

raising → starting an exception event flow

exception handling → interacting with an exception flow in some manner

unhandled exception → an exception flow that is not handled by our code

→ generally results in our program terminating abruptly

Exception Hierarchy

→ Python exceptions form a hierarchy

(we'll cover what that means precisely when we look at Object Oriented Programming - OOP)

https://docs.python.org/3/library/exceptions.html#exception-hierarchy

→ basically means that exceptions can be classes sub-divided into sub-exceptions that are more specific

→ for example a broad exception might be `LookupError`

→ more specifically it could be an `IndexError` or a `KeyError`

→ both of these are categorized more broadly as a `LookupError`

→ can choose to handle `IndexError` specifically

→ or LookupError more broadly

Exception Hierarchy

this means that if an exception object is an `IndexError` exception

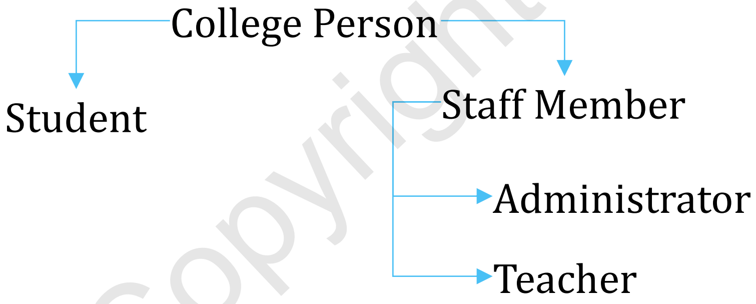
→ it is also a `LookupError` exception

→ and it is also an `Exception` exception

(most exceptions we work with are classified as `Exception` types)

→ confused? think of it this way...

Say we have these classes of objects:



→ a Teacher is also Staff Member

→ a Staff Member is also a College Person

Exception Hierarchy

→ similarly we have an exception hierarchy

```
Exception
    …
    LookupError
        IndexError
        KeyError
    OSError
        FileNotFoundError
        NotADirectoryError
        …
```

→ can even write custom exception types

→ later

→ so to handle an `IndexError`, we could choose to

→ handle `IndexError` exceptions *very specific*

→ handle `LookupError` exceptions

→ handle `Exception` exceptions *very broad*

Python Built-In Exceptions

→ Python has many built-in exception types

https://docs.python.org/3/library/exceptions.html

Common exceptions include:

→ `SyntaxError`

→ `ZeroDivisionError`

→ `IndexError`

→ `KeyError`

→ `ValueError`

→ `TypeError`

→ `FileNotFoundError`

→ and many more…

EAFP vs LBYL

→ when we think something unexpected may go wrong in our code

→ figure out if something is going to wrong before we do it

→ LBYL Look Before You Leap

→ just do it, and handle the exception if it occurs

→ EAFP Easier to Ask Forgiveness than Permission

generally in Python → follow EAFP

→ exception handling

Why EAFP?

Something that is exceptional should be infrequent

→ if we are dividing two integers in a loop that repeats 1,000 times

→ out of every 1,000 times we run, we expect division by zero to occur 5 times

LBYL → test that divisor is non-zero 1,000 times

EAFP → just do it, and handle the division by zero error 5 times

→ often more efficient

→ also trying to fully determine if something is going to go wrong is a lot harder to write than just handling things when they do go wrong

Exception Handling Flow

→ an exception occurs

→ an exception object is created

→ an exception flow is started

→ we do nothing about it

→ program terminates

→ we intercept the exception flow

→ try to handle the exception in some sense, if possible

→ then

→ resume running program uninterrupted

→ or, let the exception resume

→ or, start a new exception flow

Raising Exceptions

→ often we want to start an exception flow ourselves

→ called raising an exception

→ an exception object is associated with an exception flow

→ we create a new exception object

→ we raise the exception object

→ doing this is most useful when we create functions

→ we'll see this later

→ for now, we'll just learn how to raise an exception

Example

→ create an exception object using one of Python's built-in exceptions

```
ex = ValueError()
```

→ usually we include a custom exception message

```
ex = ValueError('Name must be at least 5 characters long.')
```

→ we raise the exception, starting an exception flow

```
raise ex
```

→ often do both in one step

```
raise ValueError('custom message')
```

→ raising an exception ourselves results in the same exception flow that Python does when it raises some exception

→ we can choose to handle the exception

→ if we don't handle the exception, program terminates

Coding

Handling Exceptions

General Suggestions for Exception Handling

→ in general we do not want to just handle any exception anywhere in our code

→ too much work

→ cannot anticipate every point of failure

→ it's OK for program to terminate - we can figure out what went wrong and attempt to fix it later – possibly handling that case specifically

→ if we don't know exactly why or where the problem occurs in our code, there's not much we can do to recover from the exception

→ we handle exceptions that are raised by small chunks of code

→ we try to handle very specific exceptions, not broad ones

→ usually handle exceptions that we can do something about

try…except…

→ wrap the code we want to implement an exception handler for inside a `try` block

→ we handle possible exception(s), using `except` blocks (one or more)

```
a = 1
b = 0
try:
    result = a / b
except ZeroDivisionError as ex:
    print(f'Exception occurred: {ex}')
    result = 0

print(result)
```

this gives us the exception object that was raised – we can assign it to any symbol we want – here I just chose `ex`

do something to handle the exception

because we handled the exception, the flow was interrupted and our code continues to run normally

Handling and re-raising an exception

→ sometimes we want to handle an exception, but then re-raise the same exception or a different exception

→ often because there's nothing we can do

→ sometimes to create a more explicit exception

to raise an exception in an `except` block:

`raise` → re-raises the same exception that caused the except block to be entered

`raise SomeException('…')` → raises a new exception

Application

→ one very common case for re-raising exceptions is for error logging

→ we can view the logs after our program has terminated abnormally

```
try:
    …
except Exception as ex:
    log(ex)
    raise
```

→ we intercept a broad range of exceptions by handling `Exception`

→ we log the exception somewhere (console, file, database, etc)

→ we re-raise the exception and let something else either handle it or terminate the program

→ what do I mean "something else handles it"?

→ we'll see this more when we cover functions

→ try…except… can be nested

→ usually indirectly

→ but directly too

```
try:
    try:
        raise ValueError('something happened')
    except ValueError as ex:
        log(ex)
        raise
except Exception as ex:
    print(f'ignoring: {ex}')
```

→ here our `ValueError` gets handled twice!

Handling Multiple Exception Types

→ not limited to a single `except` block

```
try:
    …
except IndexError as ex:
    …
except ValueError as ex:
    …
except Exception as ex:
    …
```

→ Python will match the exception to the first type that matches in sequence of except blocks

→ so write except blocks from most specific to least specific exception types

→ remember that exception hierarchy we looked at!

The `finally` Clause

→ sometimes we want some code to run after a `try…except…` whether an exception occurred or not, and whether it was handled or not

→ use the `finally` clause

```
try:
    …
except ValueError as ex:
    …
except IndexError as ex:
    …
finally:
    # always runs no matter what, before exception flow resumes
```

Application

→ useful when we want a piece of code to always run

→ whether an exception has occurred or not

→ whether the exception was handled or not

→ whether exception was re-raised or a new one raised

```
try:
    open_database_connection()
    start_transaction()
    write_data()
    commit_transaction()
except WriteException as ex:
    rollback_transaction()
    raise
finally:
    close_database_connection()
```

Coding

Iterables and Iterators

12

→ an iterable is something that can be iterated over

→ i.e. we can take one element, then the next, then the next, until we've covered all elements

→ no specific iteration order is mandated

→ obviously a sequence type is iterable (positional ordering)

→ we saw dictionaries can be iterated over (insert order)

→ but we also so sets: iterable, but no guaranteed order of any kind

→ general idea behind iteration is then:

→ start somewhere in the collection (at the beginning if that means something)

→ keep requesting the next element

→ until there's nothing left (exhausted)

→ so we have two concepts here

→ a collection of objects that we can iterate over

→ an iterable

→ something that is able to give us the next element when we request it

→ an iterator

→ we are going to look at those in this section

Iterables and Iterators

→ an iterable is something that can be iterated over

→ but we still need something that can

→ give us the next item

→ keep track of what it's given us so far (so it does not give us the same element twice)

→ informs us when there's nothing left for it to give us

→ this is called an iterator

→ used by Python to iterate over an iterable

→ an iterable is just a collection of objects

→ it doesn't know anything about how to iterate

→ however it knows how to create and give us an iterator when we need it

→ iterables implement a special method `__iter__()` that returns a new iterator

→ can also be called by using the `iter()` function

→ the iterator has a special method called `__next__()` that can be called to get the next element

→ can also use the `next()` function

→ it keeps track of what it has already handed out

(so iterators are kind of one time use!)

→ it raises a `StopIteration` exception when `next()` is called if there's nothing left

The Internal Mechanics of a `for` Loop

When we write a `for` loop that iterates over an iterable, what Python is actually doing this:

```
l = [1, 2, 3, 4, 5]

iterator = iter(l)
try:
    while True:
        # return next(iterator) - here we'll just print it
        print(next(iterator))
except StopIteration:
    # expected when we reach the end
    #   so silence this exception
    pass
```

→ the key thing here is that we can see the iterator has some state

→ it has a `__next__()` method

→ but there's no going back, or starting from the beginning again

→ to do that we have to request a new iterator

→ and that's what a `for` loop does – it requests a new iterator from the iterable before it starts looping

→ objects such as lists, tuples, string, dictionaries, sets, range objects are iterables

→ but some objects in Python are iterators – not iterables

→ iterators actually implement an `__iter__` method

→ but they just return themselves (with their current state), not a new iterator

→ they allows us to iterate over them

→ but only once

Coding

Generators

→ we've seen list, dictionary and set comprehensions

→ but no tuple comprehensions…

```
result = [i ** 2 for i in range(5)]
```

```
result = []

for i in range(5):
    result.append(i ** 2)
```

→ works because list is mutable

→ tuples are not mutable

→ no tuple comprehension

so what does this (valid) expression do?

```
(i ** 2 for i in range(5))
```

→ creates a generator object

→ generators are iterators → `next( )`

→ they calculate and hand out elements one at a time as requested

→ unlike `[i ** 2 for i in range(5)]`

→ calculates all the elements and creates the list immediately

→ generators use lazy iteration

→ a lazy property is one that is not calculated until it is requested

Why use generators?

→ memory efficiency

→ e.g. take all the rows from a file, and write them out, transformed to some other file

→ read the entire file in memory, iterate through that and save rows

→ entire file in memory!

→ you may not have enough memory!

→ read lines one at a time from file

→ read a row, process it, save it, discard it, request next row, …

→ only one line in memory at any point

Why use generators?

→ performance (possibly)

→ if you only need to read the first few elements of the iterable

→ why go through the computations to calculate all of them?

→ plus unnecessary memory usage on top of that

What's the downside of generators?

→ generators are lazy iterators

→ one-time use

→ not good if you need to iterate through the same iterable many times

→ or even just a few times if the calculations are computationally expensive or take a long time (maybe IO bound)

Creating Generators

→ use generator comprehension

→ use the yield keyword in functions instead of return

   → beyond scope of this course

Coding

Functions

13

→ we have used functions a lot so far

```
print()
iter()
next()
list()
math.sqrt()
```

and many more…

→ we can create our own, custom, functions

Why?

→ easy code re-use

→ much easier to code the `sqrt( )` function once

→ and then call it multiple times

→ breaking up complex code into easier to understand chunks

→ problem decomposition

→ when we create a function, we may also want values to be passed into it when it is called

→ arguments or parameters

→ technically not the same thing, but almost everyone uses them interchangeably → as do I ☹

when we define a function we may define symbols for the values that will be passed to the function → these symbols are called parameters

when we call a function we specify values for these parameters → these values are called arguments

→ so a parameter is when we define the function

→ an argument is when we call the function

Functions are Python Objects

→ just like everything in Python, functions are objects

→ they have state

→ name (maybe!)

→ code

→ parameters

→ they are callable → and always return something when called

→ they can be assigned to a symbol

→ can be passed as a parameter to another function

→ can be returned from a function call

Callables

→ an object is callable if it can be called – using `( )`

→ functions are run by calling them `print('hello')`

`math.sqrt(4)`

→ but other types of objects are also callable

→ not necessarily a `function` object

`my_list.copy()` → calling a method on the `my_list` object

`range(100)` → creating a new `range` object

→ more general term is a callable

Custom Functions

→ functions can be defined using the `def` keyword

`def` keyword indicates a function is being defined

function's name can be any valid Python name (just like variables)

```
def function_name():
    # indented block
    …
    return <value>
```

this block is called the function body

functions always return some value

→ function body contains any valid Python code

→ this creates a function object

→ the function object is associated with the symbol `function_name`

(in the same way `a = 10` associates the integer object `10` with the symbol `a`)

Example

```
def say_hello():
    print('Hello!')
```

`say_hello()` → `Hello!`

`say_hello()` → `Hello!`

but no `return`?

→ if a return value is not specified, function will return `None`

Example

```
def one( ):
    return 1
```

function returns the value `1` when it is called

```
result = one( )
```

→ assigns the return value of calling `one( )` to the symbol `result`

→ usually functions contain a little more complex code

```
from datetime import datetime

def current_time_utc():
    return datetime.utcnow().isoformat()


result = current_time_utc()
```

result → "2020-03-31T02:44:38.490923"

→ functions are usually more helpful when we can pass values to them

$\underbrace{\texttt{len(my\_iter)}}_{\text{we are passing an argument to the len function}}$

→ every time we call the `len` function we can pass a different value

→ the function body (implementation) of the `len` function starts running

→ it is aware of the value that was passed to it

→ same with custom functions

→ need to specify the parameters, by name, that will be used when we call it

```
def add(a, b):
    return a + b
```

`add(2, 3)` → `5`

`add(10, 1)` → `11`

```
def subtract(a, b):
    return a - b
```

`subtract(10, 7)` → `3`

→ when we call `subtract(10, 7)`, how does Python assign `10` to the symbol `a`, and `7` to `b`?

→ it does this by position

```
def my_func(a, b, c, d):      my_func(10, 20, 30, 40)
    …
```

→ positional arguments

Namespaces

→ when a function is called

→ it knows nothing about how it was called before

→ every time a function is called

→ an empty dictionary is created

→ populated with any arguments passed in

→ key = param name, value=argument

→ nothing else

→ then the function code runs

→ this dictionary is called the (local) namespace

```
abs_max(1, -2)
```

```
def abs_max(a, b):
    abs_a = abs(a)
    abs_b = abs(b)
    if abs_a > abs_b:
        max_val = abs_a
    else:
        max_val = abs_b
    return max_val
```

```
{'a': 1, 'b': -2}
{'a': 1, 'b': -2, 'abs_a': 1}
{'a': 1, 'b': -2, 'abs_a': 1, 'abs_b': 2}

{'a': 1, 'b': -2, 'abs_a': 1, 'abs_b': 2,
'max_val': 2}
```

→ after function `return`, dictionary is wiped out

→ consecutive calls to the same function are independent of each other

Coding

* Arguments

→ saw how to specify positional parameters in a function

```
def average(a, b, c, d):
    return (a + b + c + d)/4
```

→ but what if we wanted to specify an arbitrary number of parameters?

→ we'd like to call our function with different number of args

```
average(1)

average(1, 2, 3)

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

→ could write a function to use an iterable as a single argument

```
def average(iterable):
    return sum(iterable) / len(iterable)
```

→ but this makes the calling syntax a little weird

```
average([1, 2, 3])
average([1])
```

→ would be nicer if we had a mechanism to accept a variable number of args

→ Python supports a special parameter type for this

→ uses a * prefix on a parameter name

```
def average(*values):
    # return average
```

→ this means we can call `average` with any number of arguments

```
average(1)

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

→ how do we access these values inside the function

→ use the parameter name → `values` in this case

→ it will be a `tuple` containing all the argument values

```
def average(*values):
    print(type(values))
    print(values)
```

`average(1, 2, 3)` → `values` will be a `tuple`

→ `(1, 2, 3)`

```
def average(*values):
    return sum(values) / len(values)
```

→ we may want to do something if someone calls this function with no arguments

→ often you will see code that uses `*args`

→ the `*` is the important part

→ there is nothing special about the name `args`

→ as we just saw, we can use any valid name

→ use a meaningful name → `args` is often too generic

Coding

Default Values

→ possible to specify optional parameters

→ means function can be called without passing in the argument

→ but we still have that parameter

→ it needs a value

→ we can specify a default value to use if the argument is not supplied

```python
def func(a=1):
    print(a)
```

a default value to use if `a` is not supplied when function is called

`func()` → `1`

`func(10)` → `10`

→ once you specify a positional parameter with a default value

→ all positional parameters after that must specify a default value too

→ with the exception of a starred parameter

→ how would you interpret this?

```
def func(a=1, b):
    pass

func(10)
```

→ is 10 supposed to go into a? → in which case we're short one argument

→ or use default for a and assign 10 to b?

→ don't know!

→ so once we have default arguments we need to specify default for all parameters after it

```
def func(a, b, c=1, d=2):
    …
```

but this is still ok: `def func(a, b=1, *args)`

```
func(10)             a → 10   b → 1   args → (,)
func(10, 2)          a → 10   b → 2   args → (,)
func(10, 2, 3, 4)    a → 10   b → 2   args → (3, 4)
```

Coding

Keyword-Only Arguments

→ we saw how positional parameters can be passed

→ positionally

→ as a named argument → also called a keyword argument

```
def func(a, b, c):
    …

    func(1, 2, 3)

    func(c=3, b=2, a=1)
```

→ passing argument as a keyword argument is optional

→ can also make passing an argument by name mandatory

→ these are called keyword-only arguments

→ keyword-only parameters must come after all positional parameters

```
def func(a, b, c)
```

→ we want `c` to always be passed as a named argument

→ somehow we have to tell Python that after `a` and `b` there are no more positional arguments

→ one way is to use a `*` parameter

```
def func(a, b, *args, c)
```

→ since `*args` will scoop up every remaining positional argument

→ `c` must be a keyword-only argument

```
func(10, 20, 30, c=100)
```

`a` → `10` `b` → `20` `args` → `(30, )` `c` → `100`

```
func(10, 20, c=100)
```

`a` → `10` `b` → `20` `args` → `( , )` `c` → `100`

→ but this allows someone to pass in as many positional arguments as they want

→ what if we don't want that?

want to make this allowed: `func(10, 20, c=100)`

but not this: `func(10, 20, 30, 40, c=100)`

→ we still have to tell Python that there are no more positional arguments

→ we use a * without a parameter name

```
def func(a, b, *, c):
    …
```

→ a and b are positional parameters

→ there are no more positional parameters after that

→ so c is a keyword-only argument

```
func(10, 20, c=100)
```

✓

```
func(10, 20, 30, c=100)
```

✗

```
def func(a, b, *, c):
    …
```

→ using this technique `c` must be passed as a named argument

→ `a` and `b` can be passed as positional arguments

  → or as named arguments

```
func(b=2, a=1, c=3)

func(a, c=3, b=2)
```

Default Values

→ can also assign default values to keyword-only arguments

```
def func(a, b, *, c=100):
    …
```

→ `c` is optional, and will default to `100`

→ if `c` is passed, it must still be passed as a named argument

```
func(10, 20)          c → 100

func(10, 20, c=30)    c → 30
```

→ can mix default values for both positional and keyword-only arguments

Arbitrary Number of Keyword-only Parameters

→ saw `*` for arbitrary number of positional arguments

→ use `**` for arbitrary number of keyword-only arguments

```
func(a, b, *args, c, d, **kwargs)
```

→ `a` and `b` are positional

→ `c` and `d` are keyword-only

→ extra positional arguments are scooped up into `args`

→ extra named arguments are scooped up into `kwargs`

→ ** keyword-only arguments are scooped up into a dictionary

→ key is the argument name

→ value is the argument value

```
def func(a, *, d, **others):
    …
```

→ `others` is a `dict`

```
func(10, d=2, x=10, y=20)

func(a=10, d=2, x=10, y=20)

func(x=10, y=20, d=2, a=10)
```

```
a → 10

d → 2

others → {
    'x': 10,
    'y': 20
}
```

Coding

Lambda Functions

→ lambda functions are just functions

- → they are not defined using a `def` and block of code
- → it is an expression that returns a function object
  - → Python does not create a symbol or a name for the function
  - → just returns the function object
  - → we can assign it to a variable or pass it as an argument
  - → also called anonymous functions
  - → they are very simple functions (no code block)

$$\texttt{lambda}\ \underbrace{\texttt{a, b}}_{\text{function parameters}}:\ \underbrace{\texttt{a + b}}_{\text{what the function should return}}$$

function parameters what the function should `return`

→ must be a single expression

→ no code block

→ so no loops, try…except…, if…else…, etc

→ this expression returns a `function` object

→ we need to assign it to a symbol if we want to use it

```
f = lambda a, b: a + b
```

`f(10, 20)` → `30`

→ can always use a function defined using `def` instead of these lambdas

→ generally used to write shorter code in some simple cases

→ we'll see example of this in the next sections

→ but you don't have to use them

→ however they do get used often, so you should be aware of them

Coding

Some Built-In Functions

14

→ in this section we are going to look at some more of Python's built-in functions

- → there are many more!
  - → https://docs.python.org/3/library/functions.html

- → and that does not even include the thousands of functions available in Python's standard library
  - → we'll study some of them later in this course
    - → math and stats
    - → time and datetime
    - → csv
    - → random

and more…

Rounding

→ round() is a built-in function that can be used to round floats

→ uses banker's rounding

→ also called round half to even

→ rounds away from zero 1.8 → 2

-1.8 → -2

→ ties round to closest even digit 1.5 → 2

2.5 → 2

→ good choice to eliminate various biases

→ use `round( )` to round to an integer

```
round(1.8)    → 2

round(-1.8)   → -2

round(1.5)    → 2

round(2.5)    → 2
```

→ can also use round() to round to closest multiple of $\frac{1}{10}$

```
round(value, exponent)
```

`exponent` is used to specify what power of $\frac{1}{10}$ to round to

→ let's look at it mathematically first (i.e. without worrying about float representations)

`round(x, 1)` → rounds to nearest 0.1 (10^{-1})

`round(x, 2)` → rounds to nearest 0.01 (10^{-2})

`round(x, -1)` → rounds to nearest 10 (10^{1})

`round(x, -2)` → rounds to nearest 100 (10^{2})

round to closest multiple of:

`round(127.1892, 3)`	10^{-3} → 0.001	→	127.189
`round(127.1892, 2)`	10^{-2} → 0.01	→	127.19
`round(127.1892, 1)`	10^{-1} → 0.1	→	127.2
`round(127.1892, 0)`	10^{0} → 1	→	127.0
`round(127.1892, -1)`	10^{1} → 10	→	130.0
`round(127.1892, -2)`	10^{2} → 100	→	100.0
`round(127.1892, -3)`	10^{3} → 1000	→	0.0

Rounding Ties in Floats

→ technically rounds to closest number that ends with an even digit

`round(0.125, 2)` → `0.12`

→ so why this?

`round(0.325, 2)` → `0.33` why not `0.32`?

→ remember floats do not have (in general) an exact representation!

`0.325` → `0.325000000000000011102230246252`

→ so this is **not** a tie!

Coding

`sorted`, `min` and `max`

Sorting Numbers

→ numbers have a natural sort order

→ they can be sorted ascending or descending by that sort order

→ `sorted` is a built-in function that can be used to sort a collection of numbers

→ single positional argument: an iterable containing the numbers

→ by default, it sorts in ascending order

→ keyword-only argument to `reverse` the sort order

→ default is `False` → sorts ascending

→ specify `reverse=True` → sorts descending

→ always returns a new `list`

→ original iterable is not mutated

```
t = (1, 10, 2, 9, 3, 8)

sorted(t)
    → [1, 2, 3, 8, 9, 10]

sorted(t, reverse=True)
    → [10, 9, 8, 3, 2, 1]
```

Sorting Strings

Numbers have a natural sort order

Strings also have a natural sort order in Python

→ lexicographic order

→ dictionary order, alphabetical order

BEWARE The characters `a` and `A` are not the same

→ Python assigns a numerical character code (the unicode character code) to each character in a string

```
A → 65    Z → 90
a → 97    z → 122
```

→ `'A' < 'Z' < 'a' < 'z'`

→ so Python will use "alphabetical" sorting, but upper case letters will be sorted before their equivalent lower case versions

→ natural sort order of string is case sensitive

```
sorted(['Boy', 'baby'])
```

→ `['Boy', 'baby']` (ascending sort)

Sorting Other Types

→ we can "visually" sort other types of objects

→ list of Persons

→ we can sort this list

→ by name

→ by age

→ by profession

→ we always sort by some property of the objects we are sorting

→ we'll come back to this in a later section

`min` and `max`

→ closely related to sorting

→ to find the minimum of a collection

→ sort the collection (by something) in ascending order

→ pick the first element

→ to find the maximum of a collection

→ sort the collection (by something) in descending order

→ pick the first element

(or you could sort in the other direction in both cases and pick the last element) `min([1, 10, 2, 9, 8])` → `1`

`max([1, 10, 2, 9, 8])` → `10`

`min([])` → `ValueError` exception

→ can specify a `default` value to return if the iterable is empty

→ keyword-only argument

`min([], default=0)` → `0`

→ can also use an arbitrary number of positional arguments instead

```
min(1, 10, 2, 9, 3, 8) → 1

max(1, 10, 2, 9, 3, 8) → 10
```

→ we'll come back to `min` and `max` when we look at sorting again later in this course

Coding

The `zip()` Function

→ the `zip( )` function is a very useful and often used function

→ consider these two lists that contain related information

```
l1 = ['a', 'b', 'c', 'd', 'e', 'f']
l2 = [97, 98, 99, 100, 101, 102]
```

→ we want to create a list of tuples that contain the corresponding elements from `l1` and `l2`

→ could do this:

```
combo = [(l1[i], l2[i]) for i in range(len(l1))]
```

```
combo → [('a', 97), ('b', 98), ('c', 99),
          ('d', 100), ('e', 101), ('f', 102)]
```

but, we may have an issue if the two lists are not of the same length

→have to stop at the shortest of the two lengths

```
l1 = ['a', 'b', 'c', 'd', 'e']
l2 = [97, 98, 99]
```

```
combo = [(l1[i], l2[i]) for i in range(min(len(l1), len(l2)))]
```

```
combo → [('a', 97), ('b', 98), ('c', 99)]
```

→ that's what the `zip( )` function does!

```
l1 = ['a', 'b', 'c', 'd', 'e']
l2 = [97, 98, 99]

combo = zip(l1, l2)
```

BEWARE `zip( )` returns an iterator

→ remember those? → can only iterate through them once

```
list(combo) → [('a', 97), ('b', 98), ('c', 99)]

list(combo) → []
```

→ if you want to iterate multiple times over the same zipped collection

→ store it into a list

```
combo = list(zip(l1, l2))
```

→ often don't need to

`zip( )` does not actually create anything other than an iterator

→ no physical space has been used for the tuples

→ iterating over `zip( )` result, just iterates over the iterables simultaneously

→ costs almost nothing calling `zip(l1, l2)` multiple times

→ zip is extensible

→ not limited to two iterables

→ any number of iterables (positional args)

```
l1 = [1, 2, 3]
l2 = [1, 2, 3, 4, 5]
l3 = [1, 2, 3, 4, 5, 6, 7]
```

`zip(l1, l2, l3)` →

```
(1, 1, 1)
(2, 2, 2)
(3, 3, 3)
```

→ always returns an iterator that produces tuples

Coding

Higher Order Functions

15

in Python any object can:

→ be passed to a function as an argument (or callable in general)

→ be returned from a function (or callable in general)

→ functions are objects

→ functions can be passed to and/or returned from functions

→ these are called higher order functions

it's a math concept too – often referred to as operators or functionals

(functions that do not allow passing a function to or returning a function are called first order functions) amongst other things:

→ a function definition can itself contain another function definition

→ and can return it

This means we can call a function that builds another function and runs it, or even returns it

→ what becomes interesting is that variables in the outer function become available to the inner function

```
def say_hello(first_name, last_name):
    def assemble_name():
        return ' '.join([first_name, last_name])

    return ' '.join(['Hello, ', assemble_name(), '!'])
```

```
say_hello('Eric', 'Idle') → Hello, Eric Idle
```

→ we are going to study this in this chapter, along with higher order functions

→ in subsequent chapters we'll look at some important fundamental applications

Passing and Returning Functions

Passing Functions as Arguments

→ function arguments can be functions

→ the object is passed, not called

→ so don't use ( ) to pass a function, that would pass the result of the function!

```
def add(a, b):
    return a + b
```

this argument is going to receive a function object

```
def apply(func, a, b):
    result = func(a, b)
    return result
```

we now call `func`

which is whatever function was passed in

```
apply(add, 2, 3)
```

→ pass the `add` function to `apply`

→ 5

Nested Functions

→ function bodies can contain any valid Python code

→ including defining functions

```
def say_hello(name):
    def prefix():
        return 'Hello, '

    msg = prefix() + name
    return msg
```

we are creating a new function `prefix`

calling `prefix()`

Returning Functions

→ a function can also return a function

passing in a function

```
def identity(func):
    return func
```

returning the same function

```
def add(a, b):
    return a + b
```

`f` is now a symbol pointing to `add`

```
f = identity(add)

f(2, 3) → 5
```

→ silly example!

Returning Functions

→ often we return a nested function

```
def generate_func(name):
    def add(a, b):
        return a + b

    def mult(a, b):
        return a * b

    if name == 'sum':
        return add
    else:
        return mult
```

```
f = generate_func('sum')

f(2, 3) → 5
```

→ still a silly example!

→ we'll see real examples soon!

Coding

The `map( )` Function

→ the `map( )` function calls a specified function for every element of some iterable

→ very similar to doing something like this:

```
def my_map(func, iterable):
    result = [func(element) for element in iterable]
    return result
```

→ here we are creating a `list` that contains the function `func` applied to every element of `iterable`

→ but it creates a `list`

→ can take a lot of space if iterable is large

→ especially wasteful if we don't iterate over all the values

→ `map( )` returns an iterator

```
iterator = map(func, iterable)
```

as we iterate over that iterator:

- → Python moves to the next item in `iterable`
- → calls `func(element)`
- → returns the result
  - → less wasted space
  - → saves computations if we don't iterate over the whole list

→ equivalently we could also just use a generator expression

```
(func(el) for el in iterable)
```

Coding

Closures

in the last videos we saw that function definitions can be nested within another function

```
def outer():
    def inner():
        …
```

and we saw that we can return the inner function from the outer function

```
def outer():
    def inner():
        …

    return inner
```

but we can create variables in the outer function also, or pass arguments when we call it

```
def outer(a, b):
    c = 100

    def inner():
        …

    return inner
```

→ inner can "see" those variables

→ it even retains these values when it is returned

→ the inner function can "capture" those variables

→ this is called a closure

```python
def outer(a):
    def inner():
        return a * 10

    return inner


f = outer(2)
```

→ `f` is now the inner function that closes over `a` **with a value of** `2`

→ `a` is called a free variable of the closure `f`

→ we can call `f`   `f()` → `20`

→ but there are some rules!

→ you can always "read" a variable from the outer scope

reading `c` automatically uses the one in the outer scope

```
def outer():
    c = 100
    def inner():
        d = c * 10
        return d
    return inner
f = outer()
```

outer scope

inner scope

outer → `{'c': 100, 'inner': <function>}`

inner → closure `inner`, with `c=100`

→ but things change if we set that symbol to a value in the inner scope

```
def outer():
    c = 100

    def inner():
        c = 20
        return c * 10
```

here we are setting `c` to some value

→ Python ignores `c` from outer scope

→ creates a new symbol `c` in the inner scope

(there are ways around this, but beyond scope of this course)

```
f = outer()
```

outer → `{'c': 100, 'inner': <function>}`

inner → `{'c': 20}`

Example

```
def power(n):
    def inner(x):
        return x ** n

    return inner

squares = power(2)
```

→ call `power(2)`

→ `power` runs with `n = 2`

→ `inner` is a function that "captures" `n = 2` → a closure

→ the closure is returned

→ `squares` is the closure: function `inner` with `n=2` that takes one argument (`x`)

`squares(3)` → `9`

→ we can re-use `power` multiple times:

```
def power(n):
    def inner(x):
        return x ** n
    return inner

squares = power(2)
cubes = power(3)
```

[`inner` with `n = 2`]

[`inner` with `n = 3`]

```
squares(3) → 9

cubes(3) → 27
```

Coding

Sorting and Filtering

16

In this chapter we are going to focus on:

→ filtering iterables

→ sorting iterables

→ revisit min and max

Filtering

filtering is the selection of a subset of items based on whether some condition is true or not

→ given a list of numbers from 1 to 100, filter this list to contain even numbers only

→ can think of it this way:

```
          [1, 2, 3, 4, …, 98, 99, 100]
is_even?   F, T, F, T, …, T,  F,   T
```

→apply a function (`is_even`) to every item in the list

→ only keep items for which function returns `True`

Predicate Functions

a predicate function is simply a function of one or more arguments that returns `True` or `False`

for filtering in general:

→ given an iterable and a predicate function

→ only keep the items for which predicate function evaluates to `True`

```
l = [1, 2, -5, 6, -1, 0]
```

iterable

predicate function

```
def is_positive(x):
    return x > 0
```

filter

```
l = [1, 2, -5, 6, -1, 0]
pred = is_positive
```

$\longrightarrow$ 1, 2, 6

→ Python has a `filter` function that works exactly that way

```
filter(pred, iterable)
```

```
data = [1, 2, 3, -1, -2, 0]
```

```
def is_positive(x):
    return x > 0
```

```
filter(is_positive, data)
```

→ lazy iterator

→ can only iterate through this once

→ `1, 2, 3`

```
def is_even(x):
    return x % 2 == 0
```

```
filter(is_even, data)
```

→ `2, -2, 0`

→ can also use a predicate function created via a lambda

```
is_positive = lambda x: x > 0
filter(is_positive, data)
```

→ and often, directly inline with the call to `filter`:

```
filter(lambda x: x > 0, data)
```

Coding

Sorting

→ looked at the `sorted` function before

→ given an iterable

→ return a `list`, that has been sorted

→ but by what?

`sorted([10, 9, 3, 1, 2, 8])` → `1, 2, 3, 8, 9, 10`

→ sorting numbers is very intuitive

→ numbers have a natural sort order, and we can sort the elements based on their values

→ sorting strings was a bit more complicated

→ assign an integer value to each character, and use that to sort strings

→ we can sort the same data in different ways

`data = [3, 1, -6, -2, -4, 5]`

→ sort based on value → `[-6, -4, -2, 1, 3, 5]`

→ sort based on absolute value → `[1, -2, 3, -4, 5, -6]`

→ sort based on the second digit in the square root of the absolute value

→ maybe something more practical

→ sort a collection of objects (symbol, open, high, low, close)

→ by symbol

→ by open

→ by high - low

etc…

→ how do we sort by a different criteria

→ how do we sort arbitrary objects that may not even have a natural sort order

→ approach is similar to how filter worked

- → iterable
- → to each element in iterable, assign a value that is used to sort

```
data = [3, 1, -6, -2, -4, 5]
```

sort by absolute value:

```
3, 1, 6, 2, 4, 5
```

```
1, -2, 3, -4, 5, -6
1,  2, 3,  4, 5,  6
```

→ just like `filter` used a predicate function to calculate `True`/`False` for each element

→ `sorted` can take a `key` function as a named argument

- → `key` function returns a value for each element
- → those values have a natural sort order
- → usually numbers, but does not have to be

```
data = [3, 1, -6, -2, -4, 5]

def sort_key(x):
    return abs(x)

sorted(data, key=sort_key)
sorted(data, key=lambda x: abs(x))
sorted(data, key=abs)
```

→ the main point is that `key` is just a function that returns a value for each element of the iterable

→ `sorted( )` then uses that value to sort the items in the iterable

```
data = {'a': 300, 'b': 100, 'c': 200}
```

→ sort the keys of the dictionary based on the corresponding value

key_func(dict_key) → corresponding value

```
sorted(data.keys(), key=lambda k: data[k])
```

→ `['b', 'c', 'a']`

```
sorted(data.keys(), key=lambda k: data[k], reverse=True)
```

→ `['a', 'c', 'b']`

Coding

`min` and `max`

→ previously we saw that to get the minimum of an iterable

→ sort iterable from low to high

→ take first element

→ similarly with maximums

→ but we just saw that sorting always uses an associated key

→ so when we talk of `min` and `max`

→ we really have the same thing - a sort key is used

→ `min(iterable, key=<func>)`

→ `max(iterable, key=<func>)`

```
data = [-1, 2, -3, 4, -5]
```

`min(data)` → `-5`

`max(data)` → `4`

but this assumes a natural sort order

(i.e. key func is an identity function – returns the iterable value as-is)

→ let's say we want the sort to be based on the absolute value

`min(data, key=abs)` → `-1`

`max(data, key=abs)` → `-5`

Coding

Decorators

17

→ decorators are a form of metaprogramming

→ they allows us to wrap functionality around an already defined function

→ without having to modify the code of the original function

leverages:

→ closures

→ functions as first class citizens (aka higher order functions)

→ re-assign any object to an existing symbol

Why are decorators useful?

→ let's use an example to understand this

→ suppose we have a program with some functions called over and over again

→ `fun1`, `fun2`, `fun3`, etc

→ every time one of those functions is called, we want to produce a log

→ maybe just print to the console that the function was called

→ we could certainly put the "logging" functionality into each function

```python
def fun1():
    print('Called fun1.')
    …

def fun2():
    print('Called fun2.')
    …

def fun3():
    print('Called fun3.')
    …

def fun4():
    print('Called fun4.')
    …
```

→ repeating the same code multiple times

→ what if we want to include date/time call was made

→ go back and edit logging code inside each function

→ 3 weeks later, oh yeah, add some timing to it too

→ go back and edit logging code inside each function

→ too much typing! → error-prone → easy to be inconsistent

→ instead want to write the logging code once

→ and "apply" it to each function we want to log

→ basically we want to build a second function that will:

→ run some code

→ execute the original function with the arguments that were passed in

→ run some code

→ return the result of the call

→ when we call `fun1`, `fun2`, etc

`fun1(10, 20)` → start timing

→ `result = fun1(10, 20)`

→ stop timing

→ log call, date/time and timing

→ return `result`

→ using some common code

`fun2(10)` → start timing

→ `result = fun2(10)`

→ stop timing

→ log call, date/time and timing

→ return `result`

Decorators

→ recall nested functions

```
def outer():
    def inner():
        …
    return inner
```

calling `outer()` → returns the function `inner`

```
f = outer()
```

`f()` → this called `inner` returned by `outer`

→ using closures, we can do this:

```
def outer(fn):
    def inner():
        print(f'Calling {fn}…')
        result = fn()
        return result
    return inner
```

```
def hello():
    return 'Hello!'
```

`fn` is a free variable, from `outer` scope

`f = outer(hello)` → `inner` function is created

→ it is a closure with `fn` pointing to `hello`

`f()` → calls `inner`, with `fn` pointing to `hello`

→ this calls `hello()`

→ and `returns` the result of that call

→ so outer can create and return a function that will execute whatever function is passed as an argument to outer

```
def outer(fn):
    def inner():
        print(f'Calling {fn}…')
        result = fn()
        return result
    return inner
```

`f = outer(fun1)` `f()` → will call fun1 (and maybe do some extra things)

`f = outer(fun2)` `f()` → will call fun2 (and maybe do some extra things)

`f = outer(fun3)` `f()` → will call fun3 (and maybe do some extra things)

- → but we also want to be able to pass arguments to fun1, fun2, fun3
    - → so pass them to inner and use those args to call fn

```python
def outer(fn):
    def inner(*args, **kwargs):
        print(f'Calling {fn}…')
        result = fn(*args, **kwargs)
        return result
    return inner
```

`f = outer(add)` → `f` is now the `inner` function closure with `fn` → `add`

→ notice that `inner` can receive any number of positional and keyword-only args

→ whatever we pass in as arguments (when we call `f( )`)

→ will be passed as-is to whatever function `fn` points to (`add` in this case)

`f(10, 20)` → calls `inner(10, 20)`, where `fn` → `add`

→ calls `add(10, 20)`

```python
def outer(fn):
    def inner(*args, **kwargs):
        print(f'Calling {fn}…')
        result = fn(*args, **kwargs)
        return result
    return inner
```

→ can think of this as a wrapper for fn

→ we call `outer(fn)` to create a new function that wraps `fn`

→ we can call that new function with arguments

→ it does it's own thing (like `print` in this example)

→ but it also executes `fn`, with whatever arguments we pass in

→ and returns the result of that call

```python
def log(fn):
    def inner(*args, **kwargs):
        print(f'Calling {fn}…')
        result = fn(*args, **kwargs)
        return result
    return inner
```

→ we actually have a very simple logger here

→ suppose we have some functions

```python
def add(x, y):
    return x + y
```

```python
def greet(name):
    return f'Hello {name}!'
```

→ we can create new functions that will perform the same task and also run the logging code

```python
add_logged = log(add)
greet_logged = log(greet)
```

→ so now, instead of calling `add`

→ call `add_logged`

→ if we need to change logging format

→ do it in just one place!

→ but we must change all calls to `add` to now be `add_logged`

→ yikes!

→ remember that Python is a dynamic language

→ we can re-assign any object to any symbol

→ instead of: `add_logged = log(add)`

→ how about: `add = log(add)`

→ the symbol `add` now points to the new function (not the original `add`)

→ which will call the original function object

→ that's the basic decorator pattern

```
def wrapper(func):
    def inner(*args, **kwargs):
        # some code here
        result = func(*args, **kwargs)
        # some code here
        return result
    return inner


def func(a, b):
    …

func = wrapper(func)
```

→ `wrapper` is called a decorator

→ this is so common

```
def func(a, b):
    …

func = wrapper(func)
```

→ there is a shorthand notation!

```
@wrapper
def func(a, b):
    …
```

→ exactly same as above

```
def func(a, b):
    …

func = wrapper(func)
```

Coding

LRU Caching

→ this is a really interesting application of decorators

→ it solves the following problem

you have some function that gets called often

→ the same set of arguments are used often

→ the function is deterministic

→ calls with the same arguments should produce the same result

→ re-calculating the function is fairly costly

→ we could use a caching mechanism

→ first time a set of arguments is encountered, calculate result

→ store result in a cache

→ subsequent calls with same arguments, recovers result from cache

→ basic idea is this:

```
cache = {}

def func(a, b, c):
    key = (a, b, c)
    if key in cache:
        return cache[key]
    # calculations here
    cache[key] = result  # add result to cache
    return result
```

→ first time we call `func` with `func(1, 2, 3)`

  → result is calculated

  → result is inserted into `cache` dictionary using the key `(1, 2, 3)`

→ next time `func(1, 2, 3)` is called, result is returned directly from `cache` dictionary

→ I will show you in code how we could try to do this ourselves using decorators

→ Python has such a decorator – the `lru_cache` decorator

LRU → Least Recently Used

→ caches should not grow indefinitely

→ so keep the `n` most recent

→ works well when most recent calls are good predictors of upcoming calls

→ can specify the cache size we want

→ `maxsize` positional argument

→ `None` means unbounded

→ otherwise specify an int to set max cache size

```python
from functools import lru_cache

@lru_cache(maxsize=20)
def my_func(a, b):
    …
```

→ uses a decorator

→ this decorator can also take arguments

→ there is a restriction

→ the arguments passed to the function must be hashable values

→ that's because they are used as a key in the cache dictionary

Coding

Text Files

18

→ opening text files

→ read data from them

→ write data to them

→ remembering to close the file when we're done!

→ a mechanism to make sure we never forget

→ context managers

What are contexts and context managers?

→ not going to cover how to create our own → but will use some

→ a context is an area of code that is entered and exited

→ it is entered by "calling" a context manager using a `with` statement

→ it is exited when the `with` code block is exited

the context manager is responsible for

→ running some code on entry

→ running some on exit

starts a context

using this context manager

```
with open_database_connection(…) as conn:
    # work with open conn
    …

# once the with block is exited,
#   the connection is automatically closed
```

context manager's job is to open a connection upon entry

and to close the connection upon exit

Reading Text Files

Opening Files

→ to read or write a text file, we first need to open the file

`open(file_path)`

path to file you want to open

can be absolute, or relative to where the Python app is running

→ need to tell Python how we want to interact with the file, the `mode` of operation

`open(file_path, mode)`

- → `r`: read-only (default)
- → `w`: write-only, create new file, or overwrite if it exists
- → `a`: write-only, create new file, or append if it exists

→ what is returned by `open( )`?

→ an object that has many methods and properties

→ `readlines( )`

→ `closed` → is file closed?

→ `close( )` → this allows us to close the file after we're done with it

→ but it is also an iterator

→ provides iteration over the individual lines in the text file

→ `next` → `for` loop etc…

→ technically we can reset the "play head", but beyond scope of this course

→ just think of it as an iterator

Closing Files

→ always close a file after you're done with it

→ releases the resource (not unlimited number of open files)

→ writes are often buffered until the file is closed

```
f = open('file path', 'w')
# write to file
f.close()
```

Closing Files

→ but what if an exception occurs while the file is open?

→ use a `try…finally…` to always close the file, no matter what

```
f = open('file path', 'w')
try:
    # write to file
finally:
    f.close()
```

→ that's one approach

`open( )` as a Context Manager

→ `open( )` is also a context manager

```
with open('file_path', 'w') as f:
    # write to file
```

→ as soon as the context exits, file is closed

→ even if an unhandled exception occurs in context block

Coding

Writing Text Files

→ same principle as reading text files

→ open file (in write mode)

→ write to file

→ close file

→ especially important for writes, since changes may be lost otherwise

→ best is to use a context manager

Write modes

→ w `open('<file_path>', 'w')`

→ creates file if it does not exist

→ overwrites (clears out) file if it already exists

→ a `open('<file_path>', 'a')`

→ creates file if it does not exist

→ appends to end of file if it already exists

Writing Text To File

→ `f.write(<some string>)`

→ writes specified string to file

→ it does not add a `\n` character automatically

→ have to do that ourselves if we need it

→ `f.writelines(<iterable of strings>)`

→ writes each string in iterable to file

→ it does not add a `\n` character automatically after each string

→ have to do that ourselves if we need it

Coding

Modules and Imports

19

What happens when amount of code becomes very large? (e.g. lots of functions)

→ sometimes our code needs to grow beyond a single file or a Jupyter notebook

→ need to break up code into multiple files

  → each file can group similar or related functionality together

  → code in one file (like a function) should be available to the other files

→ in Python these code files are called modules

  → modules can be nested within other modules

    → modules that contain other modules are called packages

→ creating packages is beyond scope of this course

→ but we should know what they are and how to use existing ones

Built-Ins

→ Python has many built-in object types and functions

→ `bool`, `int`, `float`, `str`, `list`, `tuple`, `dict`

→ `print( )`, `filter( )`, `sorted( )`, `zip( )`, `len( )`, etc

→ these are baked right into Python

→ they're always available

→ we don't have to do anything special to use them

https://docs.python.org/3/library/functions.html

Standard Library

→ Python has a lot of libraries (modules and packages) that come standard with base Python installation

→ we have to specifically tell Python we want to use them

→ we "load" them using an `import` statement

why not just load (import) everything always?

→ there is a ton of libraries

→ do you really want to load up thousands of libraries into memory for things you don't even need?

→ other reasons too, which we'll see as we work with packages during the remainder of this course

Standard Library

Python provides a huge selection of libraries (modules and packages) that cover things like:

→ numerical and math
  → math functions
  → stats functions
  → random numbers
  → Decimal objects (alternative to floats)

→ date and time

→ CSV files

→ cryptography

→ networking, internet

and many, many more…

https://docs.python.org/3/library/index.html

3rd Party Libraries

→ sometimes the standard library is insufficient or too cumbersome

→ standard library has to be as generic as possible

→ it may provide the basic building blocks to do something

→ but you may need to write a lot of functions/code to tie them together

→ many developers create libraries that leverage the standard library (or other 3rd party libraries), but provide a higher level, easier to use interface to more specialized functionality

→ sometimes 3rd party libraries have a very narrow and specific focus

→ performance or advanced functionality might be one reason

→ `NumPy` → `SciPy` → `QuantLib`

→ `Pandas` → `Matplotlib` → `scikit-learn`

3rd Party Libraries

→ we can install those libraries, and `import` them like any package

→ where do you find a list of available 3rd party libraries

→ most exhaustive source is PyPI (Python Package Index)

https://pypi.org/

→ they can be installed using `pip install` that we saw in the beginning

→ more than 220,000 libraries 😱

3rd Party Libraries

→ how to find the "good" ones?

→ read blogs, posts, books, web sites and see what other people are using

→ does it have good documentation?

→ is it still actively developed?

→ is it widely used?

→ but maybe your need is extremely specific and very niche

→ maybe something not so great is there that will work as a starting point

But don't just look for a 3rd party library for everything you write!

→ if 3rd party library is full of bugs and unsupported, life will be painful!

→ write your own code – often it is far simpler!

→ this course cannot cover all these specialized libraries

→ we will look at some

→ by the end of this course you will have a solid foundation to easily understand and use these specialized libraries

→ Official Python docs and library docs are you best source of information

→ blog posts and similar online resources can be very helpful (unless they're just plain wrong!)

→ stackoverflow is a fantastic resource for getting questions answered

https://stackoverflow.com/

→ but at some point you will need to look into the official docs

→ start now!

Basic Imports

→ Python has quite a few built-in functions and data types (classes)

https://docs.python.org/3/library/functions.html

→ built-ins are always available

→ they are essentially "pre-loaded"

→ but there's a lot more in Python's standard library

→ way too much to load everything all the time

→ even more so with third party libraries

→ so we need to load those as needed

→ all this functionality is split up into separate modules or packages

→ think of a module as a code file

→ we then load just the modules we need

Loading a Module

→ modules are just objects

  → we need to "create" the object

  → we need to assign a symbol (variable) to that object

  → we can then use the variable to reference the module object

    → which has properties, functions, other objects

the `import` statement is used to both

  → load the module (create the module object)

  → assign a symbol to the object

Example

`import math`

`math` is a module in the standard library for math related functionality

→ the `math` module has been loaded (from file)

→ and the variable (symbol) `math` is a reference to that module object

→ `math` contains many functions, such as `sqrt`

→ like with any object, we use dot notation to reach inside the object

`math.sqrt(2)` → `1.414…`

access the `sqrt` function inside that object using `.`

this symbol points to the math module object

Aliasing

`import some_module`

→ loads `some_module`

→ creates a variable of the same name that references that module

what if we want to name that symbol something else?

`import some_module as sm`

→ loads `some_module`

→ creates a variable `sm` that references the `some_module` object

Coding

Import Variants

so far we have seen two variants of the `import` statement

```
import some_module
import some_module as alias
```

→ if we want to use something inside the module, we have to use dot notation

```
fractions.Fraction(1, 2)
fractions.Fraction(1, 4)
```

→ what if we just want to use `Fraction` inside `fractions`

→ can we avoid using `fractions.Fraction` all the time?

→ yes!

→ we can import symbols from a module directly into the corresponding symbols in our code

```
from fractions import Fraction
```

→ the `fractions` module is loaded

→ but the symbol `fractions` is NOT added to our local variables

→ instead the `Fraction` symbol is added to our local variables

→ references the `Fraction` property inside the `fractions` module

```
f1 = Fraction(1, 2)
```

→ can do the same with any module

```
from math import sqrt
sqrt(2)
```

→ what if we want more than one attribute from the module?

```
from math import sqrt, pi, factorial
```

→ `sqrt`, `pi`, and `factorial` are now available as symbols in our local scope

```
sqrt(2)

2 * pi

factorial(5)
```

Coding

Dates and Times

20

Fundamental Concepts

→ time zones and UTC

→ epoch times

→ times without dates

→ dates without times

→ dates with times

→ ISO 8601 Standard

Coordinated Universal Time

→ UTC

→ sometimes still referred to as GMT (Greenwich Mean Time)

→ world standard

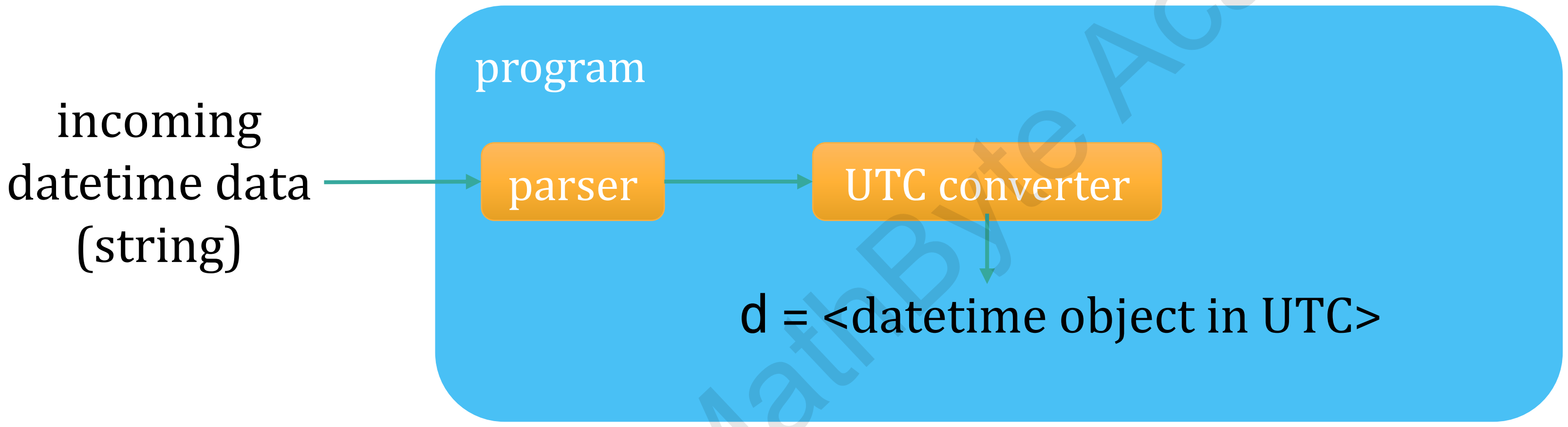
→ no adjustments for daylight saving time

→ easiest is to always use UTC internally in our programs

→ convert incoming times to UTC

→ work exclusively in UTC internally

→ display to user using their preferred time zone


incoming
datetime data
(string)
program
parser
UTC converter
d = <datetime object in UTC>

Challenges with external sources of time data

→ Python has special data types, for `time`, `date` and `datetime`

→ external sources of time data usually given as strings

→ it is a visual (string) representation of a date/time

→ but what format?

`3/1/2020 2:35:01 pm` → is this March 1, or January 3?

`March 1, 2020 14:35:01`

`03-01-2020 02:35:01 PM`

→ what time zones? → may or may not be specified, using different "standards"

→ how do we convert these to UTC based date/times in our apps?

→ what formatting should we use?

ISO Format

→ ISO 8601 defines standards for string representations of dates and times

date | time | time zone (offset from UTC)

`YYYY-MM-DD T hh:mm:ss[.nnnn]` `±hh:mm`

separator

24-hour clock
optional fractional seconds

2-digit months: `01`, `02`, `12`

2-digit days: `01`, `02`, `28`, `29`

sometimes `:` is omitted

offset, in hours and minutes (positive or negative) from UTC

optional!

`Z` is often used instead of `00:00`

→ so `Z` means UTC time zone

May 1, 2020, 10:23:35am in Eastern Time

→ daylight savings time in effect (EDT) `2020-05-01T10:23:35-04:00`

December 1, 2020, 10:23:35am in Eastern Time

→ daylight savings time NOT in effect (EST) `2020-12-01T10:23:35-05:00`

→ keeping track of all this in calculations is difficult!

→ convert to UTC first

`2020-05-01T10:23:35-04:00` → `2020-05-01T14:23:35Z`

`2020-12-01T10:23:35-05:00` → `2020-12-01T15:23:35Z`

→ then convert to whatever time zone for display purposes to user (if necessary)

→ Python has a lot of functionality for calculations with dates and times

→ to minimize introducing bugs, always use UTC based times

→ but converting these "input" times to UTC is difficult!

→ can be done using Python and the standard library

→ much easier to leverage 3^{rd} party libraries for this

→ `dateutil`
→ `pytz`

easier way to deal with:
- parsing string
- time zones

→ we'll look at these later in this course

Epoch Time

→ we saw that dealing with date/times involves time zones (whether UTC or something else)

→ introduced by Unix as a way to define a datetime without using timezones

→ start with a base datetime → the epoch

→ given a datetime, calculate it as the difference in seconds from the epoch

→ also called Unix or POSIX time

→ epoch is system dependent

→ Usually: `January 1, 1970 00:00:00 UTC`

`2020-05-01T10:23:35-04:00` → `1588343015.0`

→ but if ingesting datetime information that use epoch times, you need to know the epoch!

The `time` Module

→ used for time manipulations

→ mostly uses epoch times

→ we won't use this much

→ but it also includes some useful functions

→ `sleep`

→ `perf_counter`

The `datetime` Module

→ used for `date` (only), `time` (only) and `datetime` (date with time) objects

→ can handle time zones

→ provides formatting and parsing capabilities

→ defines a `timedelta` data type (class)

→ used to represent time difference between two date/time objects

The `time` Module

perf_counter

→ perf_counter is used to measure elapsed time in (float) seconds

→ from some undefined start (0) (usually when program starts running)

→ always look at difference between calls to perf_counter

→ uses a clock with highest available precision

```
from time import perf_counter

t1 = perf_counter()
t2 = perf_counter()
elapsed = t2 - t1
```

`sleep`

→ `sleep(n)` is used to pause execution for (float) `n` seconds

→ why would you want to slow your program down??

→ give time for something else to finish

→ usually some external resource

→ maybe a network connection is temporarily down

→ retry connecting a few times, but wait in-between retries

Getting the epoch

→ Unix systems use January 1, 1970, 00:00:00 (UTC)

`time.gmtime(n)` → returns a time object (`struct_time`)

→ based on `n` seconds elapsed from epoch

→ has the following properties:

`tm_year`, `tm_mon`, `tm_day`

`tm_hour`, `tm_min`, `tm_sec` + a few more…

→ ignores fractional seconds (if float)

→ to find the epoch on your system

```
time.gmtime(0) → struct_time(tm_year=1970, tm_mon=1,
                     tm_mday=1, tm_hour=0, tm_min=0,
                     tm_sec=0, …)
```

Getting the current epoch time

`time.time()` → returns the current time (in seconds) since the epoch

→ get UTC `time_struct` from that

```
time.gmtime(time.time())
```

Converting from `time_struct` to epoch time

→ `gmtime(n)` converts an epoch time `n` to a `time_struct`

→ can also convert a `time_struct` back to an epoch time

`calendar.timegm(time_struct)`

→ `timegm` is the inverse of `gmtime`

→ it is located in the `calendar` module

```
from calendar import timegm
n = 1_000_000_000
t = gmtime(n)    struct_time(tm_year=2001, tm_mon=9, …)
timegm(t) → 1_000_000_000
```

Formatting epoch time to human readable string

→ if we show someone an epoch time (a float), that does not mean much to them

→ as humans we are used to certain formats for the date and time

→ use the `strftime(format, time_struct)` function (string format time)

→ `format` is a `string` that contains special formatting directives

→ for example, suppose we have an epoch time: `1587253022`

(which is actually `2020-04-18 23:37:02`)

→ we can format this time into April 18, 2020 as follows:

```
t_struct = gmtime(1587253022)
strftime("%B %d, %Y", gmtime(t_struct))
```

→ `"April 18, 2020"`

here are a few more format directives:

https://docs.python.org/3/library/datetime.html#strftime-and-strptime-format-codes

`%Y` → four digit year

`%y` → two digit year

`%m` → month number

`%B` → month full name

`%b` → month abbreviated name

`%d` → day of the month number

`%H` → hour in 24-hour clock

`%I` → hour in 12-hour clock

`%p` → AM or PM

`%M` → minute number

`%S` → second number

`%z` → time zone offset `±HHMM`

`%Z` → time zone name

`%w` → weekday number (Sunday=0)

`%A` → weekday full name

`%a` → weekday abbreviated name

→ epoch time `t = 1587253022` (2020-04-18T23:37:02)

```
from time import strftime, gmtime
t_struct = gmtime(t)
```

```
strftime("%Y-%m-%dT%H:%M:%Sz", t_struct)
```

→ `'2020-04-18T23:37:02z'`

```
strftime("Today is %A, %B %d, %Y", t_struct)
```

→ `'Today is Saturday, April 18, 2020'`

```
strftime('Time: %I:%M %p %Z', t_struct)
```

→ `'Time: 11:37 PM UTC'`

Parsing Date/Time Strings

→ this is the reverse of the formatting we just saw

given a string such as: `"04/18/2020 11:37:02 PM"`

→ "convert" it to an epoch time

→ we'll assume the time was given in UTC (since no indication was given)

→ also assume format is Month/Day/Year (not Day/Month/Year)

→ in this case we can safely assume this, since there is no month 18

→ not always that lucky!

→ we need to tell Python what to expect in the string, using same directives as before

`time.strptime(date_string, format)` (string parse time)

```
from time import strptime

s = "04/18/2020 11:37:02 PM"

strptime(s, "%m/%d/%Y %I:%M:%S %p")

    → time.struct_time(
          tm_year=2020,
          tm_mon=4,
          tm_mday=18,
          tm_hour=23,
          tm_min=37,
          tm_sec=2,
          tm_wday=5,
          tm_yday=109,
          tm_isdst=-1
      )
```

→ for every date/time formatting variant, we have to specify the format to parse it

`4/18/20 23:45:34`	`"%m/%d/%y %H:%M:%S"`
`18/04/2020 11:45:34 PM`	`"%d/%m/%Y %I:%M:%S %p"`
`20/4/18 11:45:34 PM`	`"%y/%m/%d %I:%M:%S %p"`

→ this can get difficult

→ especially if our various data sources use a mixture of formats

→ this is where 3rd party libraries, such as `dateutil` can help

→ we'll come back to that later…

Coding

The `datetime` Module

→ the `time` module is a low-level library

→ good for working with epoch times

→ but a bit cumbersome

→ not a ton of functionality

→ instead use the `datetime` module

→ isolates us from epoch times (used internally)

→ provides handy data types (classes)

→ `date`

→ `time`

→ `datetime`

→ `timedelta`

→ `timezone`

datetime.date

→ date is a data type (class) for working with pure dates (no times)

```
from datetime import date
date(year, month, day)
```

(or import datetime module, and use fully qualified names)

```
import datetime
datetime.date(year, month, day)
```

→ properties:

```
.year
.month
.day
```

`datetime.date`

`date.today()`

→ returns local date as a `date` object

`<date_obj>.toisoformat()`

→ returns an ISO 8601 string for the `date` object

`date.fromisoformat("iso formatted date string")`

→ parses and creates a `date` object from an ISO formatted date string

`datetime.time`

→ `time` is a data type (class) used to work with pure times (no date)

→ it can be time zone naïve, or aware

→ `time(hour, minute, second, microsecond, tzinfo)`

→ properties: `hour`, `minute`, `second`, `microsecond`

`tzinfo` → `None` for naïve times

→ `time.fromisoformat(s)`

→ `<time_obj>.toisoformat()`

`datetime.datetime`

→ class that supports both date and time

```
datetime(year, month, day,
         hour, minute, second, microsecond,
         tzinfo)
```

→ properties for `year`, `month`, ...

→ `datetime.datetime.fromisoformat(s)`

→ `<datetime_obj>.toisoformat()`

→ `datetime.datetime.utcnow()`

→ returns naïve local date/time in UTC

Coding

Date Arithmetic

→ date arithmetic mostly involves working with

→ `dates`, `times`, `datetimes`

→ time durations (e.g. 1 day and 2 hours and 30 minutes and 15 seconds)

`datetime` module has a special class for durations

→ `timedelta`

→ subtracting one date/time from another results in a `timedelta`

→ can add or subtract a `timedelta` from a date/time

datetime.timedelta

```
timedelta(days,
          seconds, microseconds, milliseconds,
          minutes,
          hours,
          weeks)
```

→ arguments are optional and default to `0`

→ argument values are additive

`timedelta(days=1, hours=1)` → duration of 1 day and 1 hour

→ 25 hours

datetime.timedelta

```
timedelta(days,
          seconds, microseconds, milliseconds,
          minutes,
          hours,
          weeks)
```

→ notice there is no month argument

→ what does it mean to add a month to a date???

→ 31 days, 30 days, 29 days, 28 days???

1/15/2020 + 1 month → 2/15/2020?

1/31/2020 + 1 month → 2/31/2020???

`datetime.timedelta`

→ most arguments in `timedelta()` are for convenience

→ internally `timedelta` objects store the values in days, seconds, and microseconds

→ properties `.days` `.seconds` `.microseconds`

`<timedelta_obj>.total_seconds()`

→ returns the total number of seconds (fractional float) in duration

Coding

Naïve and Aware Times

→ aware time → time has a time zone attached to it

→ naïve time → no time zone info

to simplify our coding life, we made two decisions:

- → all times we create/work with will be
    - → naïve
    - → in UTC

→ the idea is that any datetime we ingest, immediately gets transformed into a naïve UTC datetime

→ convert to aware non-UTC for display or output purposes only

`timezone` Definition

`datetime.timezone` → class to define a time zone

→ name (optional)

→ UTC offset → defined as a `timedelta` object

how is that offset defined exactly?

→ time zone offset defines the number of hours and minutes that should be added to or subtracted from the corresponding UTC time

if a time zone is 4 hours "behind" UTC, then the offset is `-4 hours`

```
tz_EDT = timezone(timedelta(hours=-4), 'EDT')
```

→ pre-defined UTC timezone: `timezone.utc`

Aware `datetimes`

```
from datetime import datetime, timezone, timedelta

d1 = datetime.fromisoformat('2020-05-15T13:30:00-05:00')

tz_EDT = timezone(timedelta(hours=-4), 'EDT')

d2 = datetime(year=2020, month=5, day=13,
              hour=13, minute=30, second=0,
              tzinfo=tz_EDT)
```

Converting from one Time Zone to Another

if we have an aware `datetime`, we can easily change it to another time zone

→ use the `.astimezone(target_tz)` method of the `datetime` object

```
d1 = datetime.fromisoformat('2020-05-15T13:30:00-04:00')

tz_CDT = timezone(timedelta(hours=-5), 'CDT')

d1.astimezone(tz_CDT)
    → datetime(2020, 05, 15, 12, 30, 00, tzinfo=tz_CDT)

d1.astimezone(timezone.utc)
    → datetime(2020, 05, 15, 17, 30, 00, tzinfo=timezone.utc)
```

→ notice that the `datetime` objects remain aware

Adding or Removing Time Zone

→ careful! Do not remove time zone from a non-UTC timestamp!

→ unless you know what you are doing and this is intentional

→ ok to remove from a UTC aware timestamp – since we assume everything is UTC

→ to make a UTC aware timestamp naïve, just replace the `tzinfo` value with `None`

→ to add a time zone to a naïve timestamp, replace `tzinfo` with appropriate `timezone`

→ use the `.replace()` method on `datetime` objects

The `replace()` method

if `dt` is some `datetime` object

→ create a new `datetime` object with the exact same values:

```
dt_copy = dt.replace()
```

→ or replace one or more values while we do the copy

```
dt_copy = dt.replace(year=2021, hour=0)
```

→ in particular, we can do that with the `tzinfo` value

```
dt.replace(tzinfo=None)

dt.replace(tzinfo=timezone.utc)
```

Daylight Savings Time

→ many places change their clock twice a year – daylight savings time

→ not everyone does

→ Most parts of Arizona do not, but some do!

→ not everyone does it at the same time

→ not everyone changes by the same amount

→ when and how much has changed over the years for the same places

so, how do we convert a UTC datetime into some specific time zone?

→ it must take all these things into account → difficult!

→ Olson Database (or IANA time zone database)

https://en.wikipedia.org/wiki/Tz_database

→ the `pytz` 3rd party library → covered later

Coding

Custom Representations

→ recall the `time` module

→ `strftime()`

→ format a time struct using formatting directives

```
strftime('Time: %I:%M %p %Z', t_struct)
```

→ `strptime()`

→ parse a datetime into a struct using formatting directives

```
strptime(s, "%m/%d/%Y %I:%M:%S %p")
```

`strftime` is available for:

→ `datetime.time`

→ `datetime.date`

→ `datetime.datetime`

`strptime` is available for:

→ `datetime.datetime`

→ uses the same special formatting directives

Coding

The csv Module

21

→ earlier we saw how to read and write text files

  → we even parsed some data from a simple CSV file

→ but CSV formats vary, so more complicated than that simple example

  → often called CSV dialects

→ would require a lot of manual work on our part to deal with all these variants

→ `csv` module provides functionality to read and write a wide variety of CSV formats

  → including tab delimited, pipe (`|`) delimited

  → can deal with different line separators

    → Unix and Windows line separators are different

      → `\n` in Unix   → `\r\n` in Windows

Reading CSV Files

What is CSV Data?

CSV is a format for tabular data → rows and columns

basic idea:

→ each row in a file is a row of data

→ rows in file are separated by a newline (OS specific)

→ each field in the row is separated by a separator aka delimiter

but that brings up a few things…

→ what field separator to use? comma? → yes, but not necessarily

→ how to deal with a field containing a comma (or whatever separator)?

```
FULL_NAME,DOB,SSN
Smith, John,3/1/1985,123-456-789
```

actually a single field, but the , inside is going to cause problems

→ use some delimiter

→ maybe double quotes → but doesn't have to be!

```
"Smith, John","3/1/1985","123-456-789"
```

→ but we don't need the delimiters around the DOB or SSN

```
"Smith, John",3/1/1985,123-456-789
```

→ what if field contains the field delimiter character?

```
"Doyle, Conan","First Holmes book was the "Scarlet Letter""
```

→ double up the quotes

```
"Doyle, Conan","First Holmes book was the ""Scarlet Letter"""
```

→ or use a prefix character to "escape" the next character

→ e.g. `\` like Python (`\n`, `\t`, etc) → but doesn't have to be!

```
"Doyle, Conan","First Holmes book was the \"Scarlet Letter\""
```

→ as you can see there can be many different ways of approaching this

CSV is not a standard format

→ unfortunately CSV is not exactly a standard

→ a variety of flavors exist → dialects

→ most common one is Excel

`delimiter` (field separator) → `,`

`quotechar` (field delimiter) → `"`

doubles `quotechar` if found inside a field

only uses `quotechar` if delimiter is found inside a field

→ but these are other valid CSV formats too

```
field1|field2|field3
```
→ pipe (`|`) delimited

```
field1	field2	field3
```
→ tab delimited

tab character

Parsing CSV Data

→ default parser dialect is `excel`

  → but we can specify custom settings for `delimiter`, `quotechar`, etc

```
csv.reader(f, delimiter=',', quotechar='"')
```

optional – defaults to `"`

optional – defaults to `,`

an open file to read from

→ returns an iterator of parsed rows over the file

```
with open('some_file') as f:
    reader = csv.reader(f)  # default uses , and "
    for row in reader:
        # row is a list containing parsed fields
```

Coding

Dialects

→ in the previous lecture we saw that we can define all kinds of settings to specify CSV format

→ works, but if we need to repeat the same settings often

- → tedious typing the same code over and over again
- → error prone – might forget or mis-type one of the settings

→ instead we can bundle up all the settings into a custom dialect

- → basically just a way to package the settings once in our program
- → and re-use elsewhere in the same program multiple times

Listing Available Dialects

→ csv module comes with some pre-defined dialects

→ `excel` → `excel-tab`

→ we can add our own to that list

→ register a dialect with

→ a name for the dialect

→ values for `delimiter`, `quotechar`, etc

```
csv.register_dialect("<name>", delimiter=…, quotechar=…, …)
```

Using a defined Dialect

→ we can specify a dialect instead of individual values for `csv.reader`

`csv.reader(f, dialect='excel')` → `excel` is the default for dialect
→ same as `csv.reader(f)`

→ or we can specify our custom dialect we registered

```
csv.reader(f, dialect='my-custom-dialect')
```

Coding

More Examples

→ in this lecture we are going to process two more CSV files

→ one with some NASDAQ data (nicely formatted)

→ an older file from the US Census Bureau (oddly formatted)

Coding

Writing CSV Files

→ reverse of reading and parsing a CSV file

→ given some data, write it out to a CSV file

  → an iterable of rows

    → each row is itself an iterable of fields (columns)

→ just like reading a CSV file, we can specify formatting options

  → either using individual values (`delimiter`, `quotechar`, etc)

  → or using a dialect (built-in or custom)

→ unless there are some reasons not to, just use the standard `excel` dialect

Writing a CSV File

→ use `csv.writer`

→ then use the `writerow` method to write out each row in your data

```
with open('<file_name>', 'w') as f:
    writer = csv.writer(f, dialect='…')
    for row in data:
        writer.writerow(row)
```

→ where `data` is an iterable containing iterables of fields

```
data = [
    [row1_col1, row1_col2, …],
    [row2_col1, row2_col2, …],
    …
]
```

→ if you want a header row, write that out too using `writerow` and an iterable of headers

Coding

The random Module

22

`random` module

→ random number generators (integer, floats)

→ pseudo random

→ actually generated by an algorithm

→ gives the appearance of random number generation (uniform distribution)

→ Mersenne Twister algorithm

→ PRNG (pseudo random number generator)

→ deterministic generator

→ we know ahead of time what the sequence will be

→ not suitable for security/cryptography for example

→ but suitable for most other purposes

→ doesn't deterministic algo defeat purpose of generating random numbers?

- → goal is to generate a sequence of numbers that
  - → is uniformly distributed
  - → appears random to user

→ but if the sequence is the same every time?

- → that's a good thing when testing code
  - → testing and debugging random things is difficult
- → we can make it so the generated sequence is not the same every time program runs
- → seed value
  - → every different seed results in a different sequence
  - → use a different seed every time the program starts
  - → Python does that for us (uses the epoch time)

→ beyond uniformly distributed PRN

- → random number generator using various distributions
  (normal, lognormal, triangular, beta, gamma, and more)

- → shuffle a sequence of elements

- → random sampling

  - → without replacement → choose 5 cards from a deck of 52

  - → with replacement → roll two die (each of which can be 1-6)

    → pick two elements with replacement from {1, 2, 3, 4, 5, 6}

Interval Notation

[a, b]   a <= x <= b

(a, b)   a < x < b

(a, b]   a < x <= b

[a, b)   a <= x < b

→ [includes endpoint

→ (excludes endpoint

Random Numbers

Random seed

→ seed is used as a "primer" for different random number sequences

→ Python automatically sets one based on system time

→ so every time our program restarts we get different sequences of random numbers

→ we can override the seed value

→ useful to guarantee repeatability of "random" sequence

→ testing, debugging

`random.seed()` → uses system time

`random.seed(a)` → uses value `a` (system time if `a` is `None`)

The base PRNG

→ there is a single pseudo random number generator

`random.random()` → generates and returns the next PRN

→ `float` in `[0.0, 1.0)`

→ uniformly distributed

→ call it repeatedly to get the next number, and the next…

→ other random related functions → all use this one at their base

→ random integer generator

→ random numbers that will display certain distributions (e.g. normal)

→ shuffling, sampling

→ all use `random()`

→ all display same repeatability for same seed

Generating Random Integers

`randrange(stop)`
`randrange(start, stop, step)`

generates an integer number in
`range(stop)`
`range(start, stop, step)`

`randint(a, b)` → generates random `int` in `[a, b]`

→ equivalent to `randrange(a, b+1)`

→ syntax convenience

→ uniform distribution

→ call repeatedly to produce a sequence of random integers

Generating Random Floats

→ `random()` → random `float` in [0.0, 1.0)
→ uniform distribution

→ `uniform(a, b)` → random `float` in [a, b]
→ uniform distribution

→ `gauss(mu, sigma)` → random float
→ normal distribution
→ mean = `mu`, std deviation = `sigma`

... and more - see the online docs
https://docs.python.org/3/library/random.html

Coding

Sampling and Shuffling

Shuffling

→ in-place shuffle of items in a mutable sequence

```
l = [1, 2, 3]
shuffle(l)
```

`l` → `[3, 1, 2]`

→ `l` was mutated

Choosing a single random element

→ `choice(seq)`

→ chooses a single random element from `seq`

→ `seq` can be any sequence type (even immutable)

→ does not modify `seq` in any way

```
l = [1, 2, 3, 4, 5]

choice(l)  → 3
choice(l)  → 5
choice(l)  → 3
…
```

uniform distribution

Choosing multiple random elements at a time

→ `choices(seq, k=…)`

→ choose `k` random elements from some sequence `seq` (uniform distribution)

→ with replacements

  → the same element may get picked more than once in each set of `k` elements

→ returns result as a list of `k` elements

```
l = 1, 2, 3, 4, 5, 6      l.choices(l, k=2)   → [6, 5]
                          l.choices(l, k=2)   → [1, 3]
                          l.choices(l, k=2)   → [2, 2]
```

→ `k` can be larger than sequence length

(guaranteed to have repeated elements!)

Sampling a Population

→ `sample(population, k)`

→ population can be a sequence or a `set`, and even a `range` object

→ choose `k` random elements from some `population` (uniform distribution)

→ without replacements

→ the same element cannot be picked twice in each set of `k` elements

→ random sampling

→ `k` is the sample size

→ returns result as a list of `k` elements

→ `k` cannot exceed `len(seq)` → `ValueError` otherwise

Weighted Choices

```
l = [1, 2, 3, 4, 5, 6, 7, 8]
```

keyword-only argument

```
choices(seq, k=3)
```

→ a list of `k` random elements from `l`
→ with replacement
→ uniform distribution

→ for each pick of an element to include in the `k` choices
→ every element has the same probability of being picked

Weighted Choices

→ but we can change those probabilities

→ by specifying a sequence of weights to assign to each element of the sequence

→ if specified, `len(weights)` must equal `len(sequence)`

```
      l = [1, 2, 3, 4, 5, 6, 7, 8]
weights = [1, 1, 1, 1, 2, 1, 1, 1]

  choices(l, weights=weights, k=3)
```

→ at every pick of the `k` elements

→ 5 has two times chances of being picked than all other elements

→ weights can be floats too

→ no longer a uniform distribution

Coding

Math and Staticstics Modules

23

→ already seen `math` module

→ look at a few more functions in that module

→ `statistics` module

→ variety of simple stats

→ means, variances

→ normal distributions

`math` Module

`factorial(n)` → factorial function

`perm(n, k)` → permutations

`comb(n, k)` → combinations

`gcd(a, b)` → greatest common divisor of integers `a` and `b`

`fsum(iterable)` → floating point sum, more accurate than `sum( )`

`prod(iterable, *, start=1)` → product of all elements in iterable `dist(p, q)` → Euclidean distance between `p` and `q` (iterables)

`hypot(*coords)` → Euclidean norm of vector with specified coordinates

`sqrt` → square root

`exp(x)` → exponent (`e ** x`)

`log(x)` → natural log (base `e`)

`log10(x)` → log base `10`

`e` → Euler's constant

`degrees(x), radians(x)` → degree/radian conversions

`sin(x), cos(x), tan(x)` → trig functions

`asin(x), acos(x), atan(x)` → arc functions

`sinh(x), cosh(x), tanh(x)` → hyperbolic functions

`asinh(x), acosh(x), atanh(x)` → arc hyperbolic functions

`pi`

For full list of functions in `math` module:

https://docs.python.org/3/library/math.html

For complex number math, see `cmath` module

https://docs.python.org/3/library/cmath.html

Coding

`statistics` Module

Measures of Central Location

→ `statistics` module

→ `s` is a non-empty sequence or iterable

`mean(s)` → arithmetic average of an iterable

`fmean(s)` → converts everything to `float`, then calculates mean (faster than `mean`)

`median(s)` → median (may not be an element of the iterable)

`median_low(s)`
`median_high(s)`
ensures median is member of the iterable

`mode(s)` → applies to numeric or nominal data

Measures of Spread

`pstdev(s)` → population standard deviation

`pvariance(s)` → population variance

`stdev(s)` → sample standard edviation

`variance(s)` → sample variance

`quantiles(s, *, n=4, method='exclusive')`

→ `n=4` for quartiles, `n=100` for percentiles

→ `method='exclusive'` / `'inclusive'`

→ indicates if `s` is a sample that does/does not include most extreme population values

Normal Distribution

→ `NormalDist` data type (class)

→ used to create and manipulate normal distributions of a random variable

```
d = NormalDist(mu=0.0, sigma=1.0)

d.mean, d.median, d.mode, d.stdev, d.variance
```

`d.pdf(x)` → probability density function

`d.cdf(x)` → cumulative distribution function

`d.inv_cdf(p)` → inverse CDF (aka quantile function)

`d.quantiles(n=4)` → returns a list of `n-1` cut points for the quantiles

Normal Distribution

`d.overlap(other_normal_dist)` → calculates area overlap of two distributions

`d.samples(n)` → returns list of `n` random samples

→ supports arithmetic operations

`+` or `-` with constants → translate distribution

`*` or `/` with constants → scales distribution

```
d = NormalDist(0, 1)

d * 5 + 20
```
→ `NormalDist(20, 5)`

Normal Distribution

→ can also combine two normal distributions (+)

```
d1 = NormalDist(1, 3)        variance → 9.0
d2 = NormalDist(2, 4)        variance → 16.0

d1 + d2  → NormalDist(3, 5)
```

mean = sum of two means

1 + 2 → 3

variance = sum of two variances

9 + 16 → 25 → std dev = 5

More functionality…

→ `statistics` module has more functionality

https://docs.python.org/3/library/statistics.html#module-statistics

Coding

The decimal Module

24

→ we have seen that floats do not have exact representations

→ most of the time that's not an issue

→ often deal with transforming data

→ slight loss of precision, rounding errors, matter

→ but level of float precision is sufficient

→ you may have cases where the loss of precision is unacceptable

→ you have to store decimal numbers exactly

→ addition, subtraction, multiplication have to be exact

→ division is going to suffer from rounding errors

`1 / 3 = 0.333…` → cannot store infinite decimal numbers

→ but what precision?

→ `Decimal` objects can store decimal numbers exactly

→ but at what cost?

→ literals have to use strings to represent numbers - unwieldy

→ cannot use most `math` functions (they convert to floats)

→ many specialized math functions are defined in `Decimal`

→ arithmetic operations are slower than floats

→ they use more memory than floats

https://docs.python.org/3/library/decimal.html

→ implements IBM General Decimal Arithmetic Specification standard

http://speleotrove.com/decimal/decarith.html

Decimal Objects

The Decimal data type

→ `decimal` module → `Decimal` data type (class)

`Decimal(3)`

take the integer `3` and convert it to a `Decimal` object

`Decimal(0.1)`

take the float `0.1` and convert it to a `Decimal` object

do you see the problem here?

`0.1` is a `float` → it is already inexact, before we even pass it to `Decimal`

take the string `0.1` and convert it to a `Decimal` object

`Decimal('0.1')`

`0.1` will be stored exactly as `0.1` in the `Decimal` type

```
0.1 + 0.1 + 0.1 == 0.3
```
→ `False`

```
Decimal('0.1') + Decimal('0.1') + Decimal('0.1') == Decimal('0.3')
```
→ `True`

```
Decimal(1) / Decimal(3)
```
→ `Decimal('0.3333333333333333333333333333')`

ok to use integers (not as strings)

→ integers have exact representations

what precision?

→ default is 28 significant digits

→ we can override this value

Significant Digits

→ number of digits needed to represent the decimal number

→ leading zeros are ignored `001.2345` → `5` significant digits

→ trailing zeros are not! `1.2000` → `5` significant digits

→ important to understand how this affects arithmetic operations

`Decimal('0.15') * Decimal(2)` → `Decimal('0.30')`
(not `0.3`)

`Decimal('0.100') * Decimal('0.200')`
→ `Decimal('0.020000')`

Rounding

→ can use the `round( )` function

→ it will use a special rounding method defined by Decimal objects

```
round(Decimal('1.2335'), 3)    → Decimal('1.234')
round(Decimal('1.2345'), 3)    → Decimal('1.234')
```

→ Banker's rounding (round to closest, ties to closest even)

→ default

→ we can specify other types of rounding

Arithmetic Contexts

→ when we perform arithmetic operations on Decimal numbers

→ precision can affect results

→ rounding methodology can affect results

Example: suppose we are using a precision of 5, and Banker's rounding

```
d1 = Decimal('1.2325')
d2 = Decimal('122')
```

`1.2325 * 122 = 150.3650`

`d1 * d2` → `Decimal('150.36')`

→ only 5 significant numbers – so had to round to two decimals

→ used Banker's rounding → `150.36`

Arithmetic Contexts

→ view your current context settings

`decimal.getcontext()`

→ prec = `28`

→ rounding = `ROUND_HALF_EVEN` (Banker's rounding)

→ later we'll see how to modify the arithmetic context

IMPORTANT

→ precision of a defined Decimal number is independent of context precision

`Decimal('1.23456789')` will be stored exactly, even if context precision is `5`

→ calculations, however, will use the context precision

Mathematical Functions

→ standard arithmetic operators and functions:

`+`, `-`, `*`, `/`, `//`, `%`, `**`   `round`, `abs`, `min`, `max`, `sum`

→ careful with `math` module

you can use…   → `Decimals` get converted to `floats` first

→ `Decimal` objects implement some math functions:

```
d = Decimal('…')  → d.exp()
                  → d.sqrt()
                  → d.ln()
                  → d.log10()
```

+ more…   https://docs.python.org/3/library/decimal.html#module-decimal

Coding

Arithmetic Contexts

Arithmetic Context

→ arithmetic contexts used in decimal calculations to define many things

→ precision of intermediate calculations

→ rounding algorithm

`decimal.getcontext()` → returns the current context information

`prec` → precision (defaults to `28`)

`rounding` → the rounding algorithm (default `ROUND_HALF_EVEN`)

and more…

→ we can change those definitions

→ globally

→temporarily just for a section of code (using a context manager)

Rounding Methods

https://docs.python.org/3/library/decimal.html#rounding-modes

→ default is `ROUND_HALF_EVEN`

→ rounds to nearest, with ties to nearest even integer

`0.135` → `0.14` `0.145` → `0.14`

→ but can define other rounding methods

`ROUND_HALF_UP`

→ rounds to nearest with ties away from zero

`0.135` → `0.14` `0.145` → `0.15`

Global Context Changes

→ can modify `prec` and `rounding` in the global context

→context settings persist for the remainder of the program

```
ctx = decimal.getcontext()
ctx.prec = 5
ctx.rounding = decimal.ROUND_HALF_UP
```

IMPORTANT there seems to be an open bug in Jupyter's IPython kernel

setting the global context settings gets reset in next cell

→ temporary workaround until bug is fixed

use this as your first cell in notebook:

```
!jupyter notebook --version
```

Temporarily Changing Context Settings

sometimes we want to temporarily change the context

    perform some operations using that context

revert the context to its previous state

→ could change the global context

```
ctx = getcontext()
current_prec = ctx.prec

ctx.prec = new_prec

# perform operations

ctx.prec = current_prec
```

→ cumbersome → may even forget to switch back

Using a Context Manager

→ much easier (and safer) to use a context manager

create context and enter context manager

modify the local context

```
with decimal.localcontext() as ctx:
    ctx.rounding = decimal.ROUND_HALF_UP
    print(round(Decimal('1.12345'), 4))
```

→ Decimal('1.1235')

after exiting context manager, global context is automatically restored

```
print(round(Decimal('1.12345'), 4)
```

→ Decimal('1.1234')

Coding

Custom Classes

25

→ everything in Python is an object

- → has a type (aka class)
- → has state
- → has functionality

for example, `[1, 2, 3, 4, 5]` is an object

- → its type is `list` → we say it is an instance of a `list`
- → its state are the elements in the list
- → functionality such as `.append`

```
l1 = [1, 2, 3]
l2 = ['a', 'b', 'c']
```

- → two different objects
- → both instances of the `list` type
- → but different state

`l2.append('d')` → affects `l2`, not `l1`

Methods and Bindings

→ why does `l2.append('d')` not affect `l1`?

→ `append` is a function that works on a specific instance of the class

→ `append` is called a method of the `list` class

→ when we call the `append` method: `l2.append('d')`

→ the method is bound to the object `l2`

→ basically it will operate on `l2`

→ in general `append` will operate on whatever `list` object is specified before the dot

```
l1.append(10)

l2.append('c')
```

Custom Classes

→ we can define our own custom types (classes)

→ instances of those classes will have

→ a type (the custom type we created)

→ some state (we can store values specific to the instance)

→ functionality (methods that are functions bound to the instance)

Initializers

→ when we create a new instance of a class

→ often want to create some initial state

→ usually by passing arguments to the "creation" phase

→ this is called the initialization phase

→ creation process is started by calling the class (type)

```
a = tuple([1, 2, 3])
```

→ we are calling the `tuple` class (using `( )`)

→ passing it an argument: `[1, 2, 3]`

→ call returns a new `tuple` instance, initialized with the elements `1, 2, 3`

→ every object creation follows this basic principle

```
reader = csv.reader(f, dialect=custom_dialect)
```

→ create an instance of the `csv.reader` class by calling it

→ pass some arguments used for initialization (file and dialect)

→ call returns an initialized new instance of `reader`

```
d = Decimal('1.2345')
```

→ create an instance of the `Decimal` class by calling it

→ pass some arguments used for initialization (number string)

→ call returns an initialized new instance of `Decimal`

Classes as Blueprints

→ classes are often referred to as blueprints for creating objects

→ a single class can be used to create many instances of that class

→ each instance will have it's own state

→ the functions defined in the class become methods bound to the instance

because these functions are bound to the instance

→ **they can access the state of the instance**

suppose we have a `Person` class defined

→ we wrote our class so that the initializer requires first and last names

```
john = Person('John', 'Cleese')
eric = Person('Eric', 'Idle')
```

→ we implemented a `greet()` method to say hello

```
john.greet() → 'John says hi!'
eric.greet() → 'Eric says hi!'
```

Creating Custom Classes in Python

→ use the `class` keyword

```
class Person:
    '''A simple Person class'''
```

→ code above is as simple a class as can be

→ but Python "injects" a lot of functionality into that class for us

→ it is callable `p = Person()`

→ this created a new instance of `Person`

→ `Person` and `p` have some state Python defined for us

```
Person.__doc__ → 'A simple Person class'
Person.__name__ → 'Person'
type(p) → Person
```

and more…

Defining Classes

→ classes are like templates for creating objects

→ objects have state and functionality

→ we can define what the state and functionality is using a class

→ every instance of that class will have that functionality

→ but every instance has its own state

class → `Circle`

→ state: `radius`

→ functionality: `area( )`, `perimeter( )`

`circle_1` → `Circle(radius = 1)`

`circle_2` → `Circle(radius = 2)`

→ two different circles → each one has its own value for radius

→ but formula to calculate `area` and `perimeter` can be common

→ it just needs access to the instance value for `radius`

→ to define a class we use the `class` keyword

```
class Circle:
```

definition of class is indented

→ one (optional) part of the definition of a class is a docstring

→ basically documentation of the class

```
class Circle:
    """This class can be used to represent a circle
    and calculate area and perimeter
    """
```

→ this is a valid Python class

```
class Circle:
    """docs for class"""
```

→ class does not do much

→ but it still has quite a bit of functionality built in for us by Python

`Circle.__name__` → `'Circle'`

`Circle.__doc__` → `'docs for class'`

`Circle.__class__` → `Circle`

`Circle.__class__ is Circle` → `True`

→ Python also makes the class callable

→ `Circle` can be called to create new instances of that class

```
c1 = Circle()
c2 = Circle()
```

→ two different instances of `Circle`

`c1 is c2` → `False`

→ the type of `c1` and `c2` is Circle

`type(c1) is Circle` → `True`

→ `c1` is an instance of `Circle`

`isinstance(c1, Circle)` → `True`

→ we can set attributes directly on the instances

```
c1 = Circle()
c1.radius = 10

c2 = Circle()
c2.radius = 20
```

→ we can retrieve the attribute from each instance

```
print(c1.radius) → 10
print(c2.radius) → 20
```

→ we create instances of a class by calling the class

→ we can set/get attributes directly on the instances using dot notation

→ to create and initialize a Circle instance

```
c1 = Circle( )
c1.radius = 10
```

→ these attributes exist in the instance namespace → normally a dictionary

`c1.__dict__` → `{'radius': 10}`

→ *sometimes* the state is not in that dictionary

→ but not in this course

→ but initializing the object state this way is cumbersome

→ we'll see a better way soon!

Coding

Initialization

→ we've seen how to define custom classes

→ we call the custom class to create new instances of that class

→ but can we provide initial values when the instance is created?

→ we've seen this before!

```
d = Decimal('10.1')
```

→ creates a new `Decimal` instance

→ initialized to `10.1`

→ the initial value was passed in the same call used to create the instance

→ could mimic this initialization somewhat

```
class Circle:
    """Circle class"""


def create_circle(radius):
    c = Circle()
    c.radius = radius
    return c
```

create the Circle instance (instantiation)

set the instance radius (initialization)

return the initialized instance

```
c1 = create_circle(10)

   type(c1) → Circle
   c1.__dict__ → {'radius': 10}
```

Recall Methods

`l1 = list('abc')`
`l2 = list('def')`
two different instances of a list

`l1.append('d')`
`l2.append('g')`
same `append` function
→ but operates on two different instances of a list

`l1 → ['a', 'b', 'c', 'd']`
`l2 → ['d', 'e', 'f', 'g']`

`l1.append('d')` → `append` is bound to `l1`
`l2.append('g')` → `append` is bound to `l2`

`obj.func()` → `func` is bound to `obj`, and is called a method

The `__init__` Method

→ the `__init__` function is a special function that is called by Python when we create a new instance of a class

```
class Circle:
    def __init__(self):
        print('__init__ called…')
```

Class creation: `Circle()` does two things

→ creates a new instance of the class   let's give it some name, `new_obj`

→ calls the `__init__` function, passing `new_obj` as the first argument

→ in that sense, `__init__` is a method bound to `new_obj`

→ `__init__` is a function defined inside the class

→ but a function nonetheless

→ we can define additional parameters!

```
class Circle:
  def __init__(self, radius):
      self.radius = radius
```

→ recall what we did here

```
def create_circle(radius):
  c = Circle()
  c.radius = radius
  return c
```

→ same thing!

→ note that the name `self` is not a special name – it is just convention

→ could name it something else

→ specify this additional parameter when we create the instance

```
c = Circle(10)      c.__dict__ → {'radius': 10}
```

Coding

Instance Methods

→ create instances from classes by calling them

  → use `__init__` method to initialize instances

  → add value attributes using dot notation

→ but how do we add functionality?

```
c = Circle(10)

c.area()   →  math.pi * r ** 2
```

  → `area` needs to be a function in the class

    → bound to the instance when called with dot notation

→ exactly the same as the `__init__` function

→ define a function in the class

→ first argument will be the instance

```
class Person:
    def __init__(self, name):
        self.name = name

    def say_hello(self):
        return f'Hello, {self.name}'


p = Person('Alex')

p.say_hello()    → Hello, Alex
```

→ just like `__init__` we can pass additional parameters to methods

```
class Person:
    def __init__(self, name):
        self.name = name

    def eat(self, food):
        return f'{self.name} is eating {food.lower()}.'


p = Person('Alex')

p.eat('Broccoli')    → Alex is eating broccoli.
```

Coding

Special Methods

→ already seen `__init__`

→ provides special behavior to our custom classes

→ there are many other such methods that provide special behavior

→ they start and end with double underscores

→ often referred to as dunder methods

(so don't use this convention for your own method names!)

Object String Representations

```
l = [1, 2, 3]

print(l)      → '[1, 2, 3]'
```

a string

```
class Circle:
    def __init__(self, r):
        self.radius = r

c = Circle(10)

print(c)  → <__main__.Circle object at 0x7fc2703b4b20>
```

→ Python's default string representation of our custom objects

→ can override this default behavior

→ via special dunder methods

→ `__str__`

→ `__repr__`

`str(c)` → will call `c.__str__()`

`repr(c)` → will call `c.__repr__()`

→ why two methods?

`__str__` is used for string representation for users

`__repr__` is used for string representations for developers (more details usually)

→ `print(c)` uses `__str__` if present

→ otherwise `__repr__`

→ otherwise default (class name & object id)

Object Equality

```
l1 = [1, 2, 3]    l2 = [1, 2, 3]
```

not the same objects `l1 is l2` → `False`

but they are equal `l1 == l2` → `True`

```
class Person:
    def __init__(self, name):
        self.name = name

p1 = Person('Alex')   p2 = Person('Alex')
```

not the same objects `p1 is p2` → `False`

`p1 == p2` → `False`

→ we can override equality definition for our custom objects

→ `__eq__` method

```
class Person:
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        return self.name == other.name
```

`p1 = Person('Alex')`   `p2 = Person('Alex')`

`p1 == p2` → `p1.__eq__(p2)`

→ `True`

in general `a == b`

→ `a.__eq__(b)`

Coding

Properties

→ we have seen to define custom classes and how to

→ define instance methods

→ get/set attributes directly on the instance

```
c.radius = 10
self.radius = 10
```

sometimes called "bare" attributes

```
class Person:
  def __init__(self, name):
    self.name = name

  def say_hello(self):
    return f'Hello, my name is {self.name}'

alex = Person('Alex')
alex.say_hello()    → Hello,my name is Alex
alex.name = 'Eric'
alex.say_hello()    → Hello,my name is Eric
```

→ we have been accessing these attribute values directly

→ we have no control over what the assigned values are

→ we have no control on formatting or modifying attribute when it is read

→ sometimes we do!

we can control things in the `__init__` when the instance is created

```
class Sale:
  def __init__(self, quantity):
    if not isinstance(quantity, int):
      raise ValueError('Must be an int')
    self.quantity = quantity
```

→ cannot control how it is set subsequently

```
class Sale:
  def __init__(self, quantity):
    if not isinstance(quantity, int):
      raise ValueError('Must be an int')
    self.quantity = quantity


s = Sale(10)

s.quantity = "zero"
```

this works!

Properties

a property is like an attribute, but

→ the value is set via a method (setter)

→ the value is retrieved via a method (getter)

if `name` is a property in the `Person` class, and `p` is an instance

```
p.name = 'Alex'
```

→ calls the setter method for `name`, passing `'Alex'`

```
print(p.name)
```

→ calls the getter method for `name`, returning a value

Read-Only Properties

→ can create read-only properties

→ define a getter method

→ but don't define a setter

(write-only properties are possible, but not common, and
a little harder to achieve)

Creating a Read-Only Property

→ define a method, with the name of the property

→ decorate the method with @property

```
class Math:
  @property
  def  pi(self):
    return 3.14
```

this is a getter method

```
m = Math()
```

`m.pi` → calls the method `pi()`, bound to `m` (e.g. `m.pi()`)

```
class Person:
  def __init__(self, name):
    self._name = name
```

notice the underscore

→ convention

→ signifies _name is a private attribute to the class

→ people using this class should not modify it directly

```
  @property
  def name(self):
    return self._name
```

Read/Write Property

→ first define a getter → then define the setter

```
class Person:
  def __init__(self, name):
    self._name = name

  @property
  def name(self):
    return self._name

  @name.setter
  def name(self, value):
    self._name = value
```

all these property names must be the same

Calculated Properties

→ properties are very general

→ they are just methods

→ they do not have to be used just to return an attribute

→ they can just calculate and return some value

```
class Person:
    def __init__(self, dob):
        self.dob = dob

    @property
    def age(self):
        age = <calc current age>
        return age
```

Coding

3rd Party Libraries

26

In this and the next sections we are going to cover some popular 3rd party libraries

→ there are thousands of 3rd party libraries

→ so this is just a tiny subset

→ those libraries can have a ton of functionality

→ we can only scratch the surface in a course such as this

→ but, you will have all the tools and knowledge you need to research further

→ read the docs

→ read blog posts and see what other libraries are popular for your needs `pytz` → dealing with time zones and DST

`dateutil` → provides an "intelligent" datetime string parser

`requests` → used to query web servers and web APIs (over http(s))

`numpy` → highly efficient implementations for array processing and math computations

`pandas` → used for data manipulation and analysis

`matplotlib` → used for creating plots and charts

→ these are 3rd party libraries

→ they need to be installed

```
pip install
```

→ we already installed them at the very start of this course

→ but you can also pip install them individually

→ you need to know the package name

→ library docs will have that information

→ create a virtual environment

```
python3 -m venv env_name
py -m venv env_name
```

Linux vs Windows

→ activate virtual environment

```
source env_name/bin/activate
.\env_name\Scripts\activate
```

Linux vs Windows

→ install the library into the virtual environment

```
pip install pytz
```

The `pytz` Library

→ used for dealing with time zones

→ implements the Olson (or IANA) database

→ supports DST (daylight savings times)

→ uniform naming convention

`US/Eastern`

`America/New_York`

`Europe/Paris`

→ Area / Location

→ goes back to 1970 (Unix epoch)

→ https://pythonhosted.org/pytz/

→ `pip install pytz`

```
import pytz
```

`pytz.all_timezones` → returns a list of all named time zones

→ internally uses Python's `tzinfo`

→ but with some extras used for DST

→ a `pytz` timezone can be used instead of a `tzinfo` object

Looking up a Time Zone

→ can retrieve a time zone from its name

```
pytz.timezone('US/Eastern')

pytz.timezone('UTC')
```

→ `pytz.UTC`

→ can use these time zones instead of Python's `tzinfo`

```
datetime(
      2020, 5, 15, 10, 0, 0,
      tzinfo=pytz.timezone('US/Eastern')
)
```

Making a naïve `datetime` aware

→ use pytz time zone's `localize` method

```
tz_ny = pytz.timezone('America/New_York')
tz_ny.localize(naive_dt)
```

→ pytz will figure out if it needs to use DST or not!

→ this just attaches the time zone information to the naïve datetime

→ it does not "convert" the datetime to the new timezone

i.e. it assumes the datetime was given in the timezone that is being attached

Converting aware datetimes to other time zones

→ once we have an aware datetime we can convert it to another timezone

→ use the `astimezone` method of the `datetime` object

→ but because we are using `pytz` timezone objects, conversions work fine, including DST calculations

→ if we start with a naïve UTC time, we can directly transform it to a specific timezone

```
<py_tz_timezone>.fromutc(<naïve datetime>)
```

Coding

The `dateutil` Library

→ https://dateutil.readthedocs.io/en/stable/

→ `pip install python-dateutil`

→ `parser`

- → ability to automatically parse dates and times from string in various formats
- → this is what we'll look at in this course

but it has a lot more…

→ computing dates based on advanced recurrence formulas

- → generate sequence of dates weekly on Tuesday and Thursday for 5 weeks
- → generate sequence of dates every weekday for 3 months
  - → very similar to what you might see when you set recurring calendar meetings

Basic Parsing Functionality

```
from dateutil import parser

parser.parse('2020-01-01T10:30:00')

parser.parse('2020-01-01 10:30:00 am')

parser.parse('12/31/2020')

parser.parse('31/12/2020')
```

Ambiguous Month/Day

`4/3/2020`   `2020/4/3`

→ is this Month/Day or Day/Month?

→ parser default assumes Month/Day

i.e. month is specified first

→ can override this by using `dayfirst` keyword argument

`parser.parse('2020/4/3')` → April 3, 2020

`parser.parse('2020/4/3', dayfirst=True)` → March 4, 2020

raises a `ParserError` exception if date is invalid or unrecognizable

Fuzzy Parsing

→ parser can even attempt parsing strings that contain extra information

→ March the 4th, 2020

→ default parsing will not work

→ use `fuzzy_with_tokens=True` argument when calling `parse`

→ returns a tuple (parsed datetime, ignored text elements)

→ raises a `ParserError` exception if date is invalid or unrecognizable

→ it's quite good, but cannot handle just anything

→ May the fourth, 2020 is not recognized

Coding

JSON Data

→ JavaScript Object Notation

→ it is a simple way of representing objects using just strings

  → very easy to transmit strings

  → over a network, as a text file, etc

→ JSON is a lightweight standard that we can use to

  → encode an object into a string → serialization

  → decode a JSON string into an object → deserialization

→ most often used when transmitting data over the web (e.g. REST APIs)

→ JSON is very simple

→ easy for humans to read and write JSON

→ easy for computers to parse and generate

→ it is a pure text format

→ language independent (Python, C++, C#, JavaScript, Java, etc) consists of:

object → unordered `key:value` pairs delimited by `{ }` (dictionary)

array → ordered list elements separated by `,` and delimited by `[ ]` (list)

values → numbers (integer or with decimal point)

→ strings, delimited by double quotes `"…"`

→ boolean `true` or `false` (note the lowercase!)

→ `null` (`None`)

→ object → so objects can contain other objects, arrays

→ array → arrays can contain other arrays, objects

→ basically JSON looks like a Python dictionary!

→ a JSON object has a single root object – everything else is nested within it

Example

```
'''
{
  "firstName": "Eric",
  "lastName": "Smith",
  "address": {
    "country": "USA",
      "state": "New York",
  },
  "age": 28,
  "favoriteNumbers": [42, 3.14],
  "likesSushi": false,
  "driversLicense": null
}
'''
```

it's a string

root is an object

key: value pairs

key must be a string

strings must be double-quote delimited

value is another object

value is a list

Important: order of key:value pairs is irrelevant in JSON – don't count on it!

→ white spaces (spaces, tabs, newlines) do not matter

```
'''
{
  "firstName": "Eric",
  "lastName": "Smith"
}
'''
```

```
'''{"firstName":"Eric","lastName":"Smith"}'''
```

→ but which is more human readable?

→ note the stylistic difference: `camelCase` vs `snake_case`

→ of course, they are just strings, so you can use whatever you want

Deserializing JSON (decoding)

→ Python standard library `json` module

→ `json.loads(json_string)`

→ parses a json string and returns a `dict` object

Since JSON is a standard, Python's `loads` can handle any standard JSON object

Serializing JSON (encoding)

→ `json.dumps(dict)` → returns a JSON string

→ have to be more careful here

→ basic JSON data types are very simple: `int`, `float`, `str`, `bool`, `None`

→ Python has a far richer set of data types

→ `datetime`, `Decimal`, custom classes, etc

→ those are not serialized by default, and if we try, we'll get an exception

→ there is a way to specify custom encoders

→ beyond the scope of this course

Coding

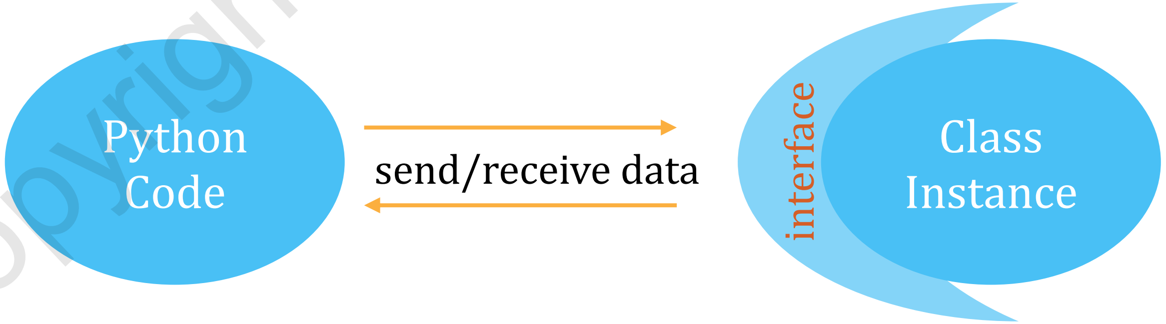
REST APIs

What is an API?

API → Application Programming Interface

→ enables your application to interact with another application

→ a Python class exposes an API → methods, properties



→ these days many applications are "in the cloud"

→ CRM → Payroll → trading platforms → Automated AI

→ they expose an API available via the web using http(s)

→ web sites → request data using a URL

→ this is called a GET request (fetches data)

How a browser retrieves a web page

`GET` `https` `://` `mysite.com` `/path/to/page.html`

method verb | protocol | domain | resource path

URI (*Uniform Resource Identifier*)

URL (*Uniform Resource Locator*)

→ also supports query arguments

→ basically like named arguments in Python functions

```
GET https://mysite.com/currentTemp?city=Chicago&units=metric
```

→ web server at mysite.com waits to receive these requests

→ browser sends request to web server

→ server sends back data (often html, but does not have to be!)

→ browser displays returned data

Sending Data

→ can also send data to a web server → e.g. user registration data

→ different methods/verbs → e.g. POST

→ specific "path" on web server we need to send the data to (specific URL)

→ data is attached when request is sent by browser

→ web server receives this data and does something with it

→ and usually returns a response of some kind

In general…

→ web servers listen for incoming requests

→ request contains

→ method GET, POST, …

→ URL → specifies exactly what we are trying to "access"

→ query arguments (maybe)

→ "attached" data (maybe)

the set of what URLs, query arguments, methods and data a web server understands

→ is essentially an API

→ data is not necessarily HTML – can be JSON, XML, …

REST APIs

→ REST APIs are special types of APIs

→ REST has to do with how they are implemented and their behavior

→ as users of the API we don't actually care if it's REST or something else!

→ one of the fundamental characteristics of a REST API is that calls are independent of each other (stateless)

→ call to API does not rely on remembering how you interacted with it in the past

→ not quite the same with web sites

→ log in

→ now you can access pages on the site

→ web server remembers who you are

→ stateful

Authentication / Authorization

→ REST APIs are generally secured

→ you need to be authenticated → web server needs to know who you are

→ usually a secret token you pass in the request

→ in something called headers

→ just an extra "bucket" of key-value data that can be sent/received along with request

→ you also need to be authorized to perform the request

→ you may be authorized to read some data

→ but you may not be authorized to create/delete that data

Authentication → establishes who you are to the system you are interacting with

Authorization → governs what you can/cannot do in system

API Data Formats

→ most modern APIs use JSON for sending/receiving data

→ sometimes uses XML, or even proprietary formats

simple, easy to read

```json
{
  "firstName": "Davey",
  "lastName": "Jones",
  "ship": "Flying Dutchman",
  "lastSeen": 2017,
  "nationality": null
}
```

more verbose, but also more powerful

```xml
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <firstName>Davey</firstName>
  <lastName>Jones</lastName>
  <lastSeen>2017</lastSeen>
  <nationality/>
</root>
```

Resources

→ REST APIs allow us to interact with entities, called resources

→ bank account
- → create new account
- → list accounts for specific customer
- → get balance
- → deposit, withdraw
- → delete the account

→ customer
- → create new customer
- → get customer info
- → update customer info
- → delete customer

```
GET https://.../customer/12345/account/5523?query=balance
```

→ `{"balance": 2123.45, "asOf": "2020-04-05T15:35:45+00:00"}`

```
POST https://.../customer/12345/account/5523
    + {"action": "withdraw", "amount": 100.0}
```

→ `{"balance": 2023.45, "asOf": "2020-04-05T16:00:00+00:00"}`

API Methods

→ since humans design/write these APIs, things are not always consistent!

`GET` → retrieves resource(s)
→ often used with query args

`POST` → used to create a resource
→ issuing the same POST request twice can end up creating two resources

`PUT`, `PATCH` → usually used for updating an existing resource

`DELETE` → delete a resource

Status Codes

→ making an HTTP request (GET, POST, etc) always returns a status code

→ plus whatever else the API specifies

2xx → success

200	→ OK	request was successful
201	→ Created	resource created successfully
202	→ Accepted	request accepted, but not finished processing (async)

4xx → you did something wrong

400	→ Bad Request	server did not understand the request
401	→ Unauthorized	technically this means "not authenticated"
403	→ Forbidden	this means not authorized
404	→ Not Found	server cannot find specified resource

5xx → Server had an issue → usually not your fault!

→ many more... https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

Finnhub Stock API

→ `https://finnhub.io`

→ provides free and paid tiers

- → REST API
- → uses JSON
- → mostly GET requests

→ you'll need to sign up for an account to follow along (free tier)

- → this is the web
- → things change often!
- → by the time you view these videos their APIs could have changed
- → but I'll show you how to read the documentation in case that happens
- → generally things stay backward compatible – clients get really annoyed otherwise!

Coding

well, almost…

The `requests` Library

→ python has a module in the standard library for making http requests

→ slightly low level interface (think time vs datetime)

→ 3rd party library   Requests: HTTP for Humans

→ pretty much standard

→ even Python's own docs suggest using it!

https://requests.readthedocs.io/en/master/

```
pip install requests
```

Making Requests

→ all standard methods/verbs are implemented as functions

```
requests.get(…)
requests.post(…)
requests.put()
```

etc…

→ common arguments

`url` → the URL request will be sent to

`params` → dictionary of query parameters (key = value)

`json` → JSON sent in request (usually for POST, PUT, etc)

`headers` → dictionary of headers (key = value)

and many more…

Receiving Responses

→ result of making a request (get, post, etc) is a `Response` object

→ it has the following properties (amongst others):

`status_code`	→ e.g. `200`, `403`
`reason`	→ e.g. `OK`, `Forbidden`
`text`	→ content of the response
`json`	→ returned deserialized JSON (if any) → so a `dict` → reading this property if no JSON is present raises a `ValueError`
`headers`	→ dictionary of headers received from server
`cookies`	→ cookies received from server

Example: Google search results (HTML response)

search:

→ search terms: python http requests

→ number of results: 5

https://www.google.com/search?q=python+http+requests&num=5

→ using requests library to retrieve the HTML search results

```
response = requests.get(
    url='https://www.google.com/search',
    params={'q': 'python http requests', 'num': 5}
)
```

```
response.status_code → 200
response.reason → OK
response.text → HTML page browser would display
```

→ calling an undefined URL

```
response = requests.get(
    url='https://www.google.com/search2',
    params={'q': 'python http requests', 'num': 5}
)
```

```
response.status_code → 404

response.reason → Not Found
```

Coding

NumPy

27

→ NumPy is a widely used library mainly used for working with arrays

- → very fast
- → very memory efficient
- → very flexible

```
pip install numpy
```

https://numpy.org/

What are arrays?

→ basically lists

→ a Python `list` is a type of array

→ elements are indexed → `arr[0]`, `arr[1]`, ...

→ array can be sliced → `arr[start:stop:step]`

→ variable size → can add / remove elements from array

→ heterogeneous → elements can have different data types

→ a NumPy array (`ndarray`)

→ fixed size

→ homogeneous

Python `list` vs NumPy `ndarray`

→ these are some of the similarities and differences

	`ndarray`	`list`
	fixed size	variable size
	can be reshaped	
	homogeneous	heterogeneous
	elements have specialized, restricted data types	elements are Python objects
indexing	`arr[i]`	`lst[i]`
slicing	`arr[a:b:c]`	`lst[a:b:c]`
masking	`arr[(arr > 2) & (arr < 10)]`	
fancy indexing	`arr[[0, 3, 4]]`	

NumPy Efficiency

→ more space efficient than Python

→ array manipulation and calculations are much faster

→ vectorization

→ but at a cost

→ fixed size → once created, cannot add/remove elements
→ elements can be replaced

→ homogeneous → all elements must be the same type
→ even in multi dimensional arrays (arrays of arrays)

→ data types → it uses data types from underlying C language
→ memory efficiency & vectorization

Integer Sizes

→ integers are stored as sequences of bits (`0`s and `1`s)

→ number of bits determines how large the integer can be

4 bits   largest number   → $(1111)_2 = 2^0 + 2^1 + 2^2 + 2^3 = 15$

→ range is: `[0, 15]` (16 numbers)

→ but may want negative numbers

→ in that case, one bit is reserved to keep track of the sign

→ 3 bits   → $(111)_2 = 2^0 + 2^1 + 2^2 = 7$

`-7 -6 … -1 -0 +0 +1 +2 … +6 +7`

→ `0` does not need a sign   → `[-8, 7]`

Integer Sizes

signed integers	→ 8 bits	[-128, 127]
	→ 16 bits	[-32_768, 32_767]
	→ 32 bits	[-2_147_483_648, 2_147_483_647]
	→ 64 bits	[-9_223_372_036_854_775_808, 9_223_372_036_854_775_807]
unsigned integers	→ 8 bits	[0, 255]
	→ 16 bits	[0, 65_535]
	→ 32 bits	[0, 4294967295]
	→ 64 bits	[0, 18_446_744_073_709_551_615]

Floats

→ Python uses 64 bits to store floats

→ certain precision and size of exponent

→ C also has 32-bit floats

→ less precision, smaller exponent

→ but more efficient storage

NumPy Types

→ in NumPy you choose your data type

→ if you pick an unsigned 8-bit integer, you can only store numbers in `[0, 255]`

signed integers → `int8`, `int16`, `int32`, `int64`

unsigned integers → `uint8`, `uint16`, `uint32`, `uint64`

floats → `float32`, `float64`

(`float64` is compatible with Python `float`)

complex → `complex64`, `complex128`

(`complex128` is compatible with Python `complex`)

https://numpy.org/doc/stable/user/basics.types.html

Vectorization

suppose we want to multiply every element of one array by the corresponding element in another array

```
a = [1, 2, 3, 4]
b = [10, 20, 30, 40]
```

→ `result = [10, 40, 90, 160]`

→ loop

```
result = []
for i in range(4):
  result.append(a[i] * b[i])
```

or `[x * y for x, y in zip(a, b)]`

at every loop, Python must:

→ lookup the operand objects

→ determine the types

→ try to perform the operation (if `a * b` does not work, it tries `b * a`)

C does not have to do all that work → significantly faster

Vectorization

NumPy implements things in such a way that

→ given `a` and `b` are NumPy arrays (`ndarray`)

→ given a supported function or operator

```
a + b              → add(a, b)
a * b              → multiply(a, b)
a / b              → divide(a, b)
sin(a) / sin(b)    → divide(sin(a), sin(b))
```

→ NumPy pushes the loop and calculations down into `C`

→ this is called vectorization

→ these functions are called universal functions (ufunc)

Why are Arrays Important?

→ most data we deal with is represented as arrays

  → often multi-dimensional arrays

→ an image is a 2-dimensional array of colored pixels

  → each pixel is an array, e.g. [red, green, blue, alpha]

→ a video is an array of images (a bit oversimplified)

→ audio is encoded into arrays

→ stock quotes, tick data are arrays of data

→ an Excel spreadsheet is a (2-dimensional) array

NumPy is a huge library

→ lots of universal functions

→ financial, math, stats, linear algebra, sorting, sampling, Fourier transforms (discrete) and more…

→ we'll just look at a few of these

→ also introductory look at array creation and manipulation

(indexing, slicing, fancy indexing, masking, reshaping)

https://numpy.org/doc/stable/

→ it also is the foundation for the Pandas library (dealing with data sets)

Creating Arrays from Lists

→ first thing is we have to import NumPy

```
import numpy
```

→ typically everyone aliases it for less typing

```
import numpy as np
```

→ the array type is `np.ndarray`

(n-dimensional array)

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

`type(a)` → `ndarray`

→ but what type was used for the elements themselves?

→ remember that in NumPy we use the C types, not the Python types

→ also array is homogeneous, i.e. every element has same data type

`a.dtype` → `int64`

→ NumPy analyzes the data and picks something appropriate

→ in this case a 64-bit integer

→ for floats it defaults to 64-bit floats

Specifying the element data type

→ we can override that default and select a specific type

```
a = np.array([1, 2, 3], dtype=np.int8)
```

`a.dtype` → `int8`

Careful!

- → do not use a type that is too restrictive
- → weird things happen when integer in list is too large for specified `dtype`
- → floats in a list will be truncated if `dtype` is set to an integer

→ why not just always use `int64`?

- → memory efficiency for extremely large datasets

Multi-Dimensional Python Lists

→ in this course we'll stick to 2 dimensional arrays

```
l = [
        [1, 0, 0],
        [0, 1, 0],
        [0, 0, 1]
    ]
```

rows

columns

→ also called axes

rows → axis 0

columns → axis 1

→ only using 2 dimensions is not particularly restrictive

time	open	high	low	close	prev_close
1603249488	100	102	98	102	100
1603249498	200	202	198	202	200
1603249587	300	302	298	302	300

Converting Multi-Dimensional Lists to Arrays

→ works exactly the same way as with 1-D arrays

→ but again, remember that all the elements in the array must be of the same type

```
l = [
        [1, 0, 0],
        [0, 1, 0],
        [0, 0, 1]
    ]
```

```
m = np.array(l)     dtype → int64

m = np.array(l, dtype=np.uint8)
```

Array Shape

→ shape of an array is number of elements in each dimension

```
[
   [1, 2, 3],
   [4, 5, 6]
]
```

→ 2 dimensions

→ first dimension has 2 elements

→ second dimension has 3 elements

→ `(2, 3)`

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

→ 1 dimension

→ first dimension has 3 elements

→ `(3, )`

→ use the `shape` attribute of `ndarray` objects

Coding

Creating Arrays from Scratch

→ seen how to create arrays from lists

→ handy to convert lists of data loaded from a CSV file for example

→ or retrieved via a web API

→ sometimes we just need to generate specialized arrays

→ could do it from a Python list

→ but NumPy has several convenient functions

Array of zeros

`np.zeros(size_or_shape, dtype)`

optionally specify data type
defaults to `float64`

single number → 1-D array of that length

tuple → shape (# rows, # columns)

```
[0, 0, 0]      [
                  [0, 0, 0],
                  [0, 0, 0]
               ]
```

`np.zeros`	→ arrays filled with zeros
`np.ones`	→ arrays filled with ones
`np.full`	→ arrays filled with some specified constant value
`np.eye`	→ generates identity matrices
`np.arange`	→ generates 1-D array based on a range (start:stop:step)
`np.linspace`	→ generates evenly spaced numbers between start/stop
`np.random.random`	→ arrays filled with random floats `[0, 1)`
`np.random.randint`	→ arrays filled with random integers `[low, high)`

Coding

Reshaping Arrays

What is reshaping?

```
[1, 2, 3, 4, 5, 6]     shape → (6, )
```

→ using the same elements, we can rearrange them

```
[
  [1, 2, 3],
  [4, 5, 6]
]

  (2, 3)
```

```
[
  [1, 2],
  [3, 4],
  [5, 6]
]

  (3, 2)
```

```
[
  [1],
  [2],
  [3],
  [4],
  [5],
  [6]
]

  (6, 1)
```

Reshaping Shares Elements

→ this is very important (and we'll see later this applies to slicing also)

```
 arr = np.array([1, 2, 3, 4, 5, 6])
shap = arr.reshape(3, 2)
```

```
                  [1, 2, 3, 4, 5, 6]
[
  [1, 2],
  [3, 4],
  [5, 6]
]
```

these are the same elements

→ if you change the value at `[0, 0]` in `shap`, it will change the corresponding value in arr `[0]` and vice versa

→ in a sense, reshaping rearranges the "slots"

Making a Copy

→ `arr.copy()`

→ this will make a copy of `arr`

→ can use to break the tie between an array and the reshaped array

Coding

Stacking

→ concept is very straightforward

→ we can stack arrays one on top of each other (`vstack`)

→ or we can stack then side by side (`hstack`)

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{bmatrix} \xrightarrow{\texttt{hstack}} \begin{bmatrix} 1 & 2 & 3 & 10 & 20 & 30 \\ 4 & 5 & 6 & 40 & 50 & 60 \\ 7 & 8 & 9 & 70 & 80 & 90 \end{bmatrix}$$

$$\begin{matrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \\ \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{bmatrix} \end{matrix} \xrightarrow{\texttt{vstack}} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{bmatrix}$$

Stacking

→ `a1`, `a2`, `a3` are arrays

```
np.vstack((a1, a2, a3))
```
→ stack vertically

```
np.hstack((a1, a2, a3))
```
→ stack horizontally

argument is a tuple

Shapes must be Compatible

→ if stacking vertically, same number of columns for each array is required
→ if stacking horizontally, same number of rows for each array is required

vstack

```
1 2 3
4 5 6
-----
7 8 9
-----
0 0 0
```

```
1 2 3
4 5 6
--------
7 8 9 10
--------
0 0 0
```

hstack

```
1 2 | 10 20 30 | 0
3 4 | 40 50 60 | 0
5 6 | 70 80 90 | 0
```

```
      | 10 20 |
1 2   | 40 50 | 0
3 4   | 70 80 | 0
5 6   | 90 99 | 0
```

What happens to `dtype`?

→ can stack arrays with different `dtype`

→ NumPy will determine a suitable common data type

→ we cannot control that

→ stacking `uint8`, `uint16` and `int64`

→ NumPy picks a `float64` for the stacked array

in a future version of NumPy (1.20), it will be possible to specify the data type when using the `concatenate` function – which is a more generic form of `vstack` and `hstack`

Casting an Array to another Data Type

→ we can however control the stacked data type by first converting the arrays we are stacking to a common type

→ use the `astype` method on an array

```
arr1.astype(np.int64)
```

→ so we could use this to stack multiple arrays

```
np.vstack(
        [
            arr1.astype(np.int64),
            arr2.astype(np.int64)
        ]
)
```

Stacked Arrays are Independent of Original Arrays

→ we saw that a reshaped array is "linked" to the original array

→ this is not the case for stacked arrays

  → modifying an element in the stack does not modify original array

  → modifying element in original array does not modify the stack

Coding

Indexing

Python Sequence Types

→ recall Python sequence types such as lists and tuples

→ elements are positionally indexed `0, 1, 2, …`

→ get element at index `i` `lst[i]`

→ replace element at index `i` `lst[i] = x`

→ indexing 2-D lists (a list of lists) works the same

```
arr = [ [1, 2], [3, 4] ]

arr[0][1] → 2

arr[0][0] = 100        arr → [ [100, 2], [3, 4] ]
```

Indexing NumPy Arrays

→ very similar to Python sequence types

```
arr = np.arange(1, 7).reshape((2, 3))
```

→

```
[
  [1, 2, 3],
  [4, 5, 6]
]
```

`arr[0][0]` → `1`

`arr[1][2]` → `6`

→ with NumPy arrays instead of `[i][j]`, we can use `[(i, j)]`

a tuple, so we can omit the `( )`

```
arr[0][0]    arr[0, 0]

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

→ for 1-D array

```
arr = np.arange(1, 7)
```

`arr[1]` → `2`

`arr[(1,)]` → `2`

Mutating Elements

→ works the same as Python lists

```
arr = np.arange(1, 7)

arr[2] = 30        arr → [1, 2, 30, 4, 5, 6]
```

```
arr = np.arange(1, 7).reshape((2, 3))  →  [
                                             [1, 2, 3],
                                             [4, 5, 6]
                                          ]

arr[1, 2] = 60  →  [
                     [1, 2, 3],
                     [4, 5, 60]
                   ]
```

BEWARE: data types!

Coding

Slicing

Slicing Python Sequences

```
l = [1, 2, 3, 4, 5]
```

`l[0:3]` → `[1, 2, 3]`

→ slicing returns a new, independent, list

```
slice_ = l[0:3]

slice_[1] = 20
```

`slice_` → `[1, 20, 3]`

`l` → `[1, 2, 3, 4, 5]`

Slicing Python 2-D Sequences

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

→ want to slice in two axes

```
m[0:2]   →   [
               [1, 2, 3],
               [4, 5, 6]
             ]
```

→ cannot just use a slice to isolate

```
[
  [2, 3],
  [5, 6]
]
```

Python Sequence Slice Assignments

→ we can mutate a Python list by using the assignment operator with a slice definition

```
l = [1, 2, 3, 4, 5]
    l[0:3] = [10, 20, 30]  l → [10, 20, 30, 4, 5]
```

→ since Python lists are not fixed size, we can also replace the slice with more or less elements

```
l = [1, 2, 3, 4, 5]
l[0:2] = [10, 20, 30, 40]   l → [10, 20, 30, 40, 3, 4, 5]
```

Slicing 1-D NumPy Arrays

→ very similar to slicing lists

`arr = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8])`

`arr[0:3]` → `[0, 1, 2]`   `ndarray` (not `list`)

→ step, negative indexing, etc are all supported, just like `list` slicing

`arr[2:6:2]` → `[2, 4]`

`arr[1::2]` → `[1, 3, 5, 7]`

`arr[::-1]` → `[8, 7, 6, 5, 4, 3, 2, 1, 0]`

Slicing 2-D Arrays

→ NumPy provides support for slicing along multiple axes

axis 1

$$\text{axis 0}\quad \begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 2 & 3 \\ 1 & 4 & 5 & 6 \\ 2 & 7 & 8 & 9 \end{array}$$

slice `0:2` along axis 0

and slice `1:3` along axis 1

```
arr[0:2, 1:3]
```

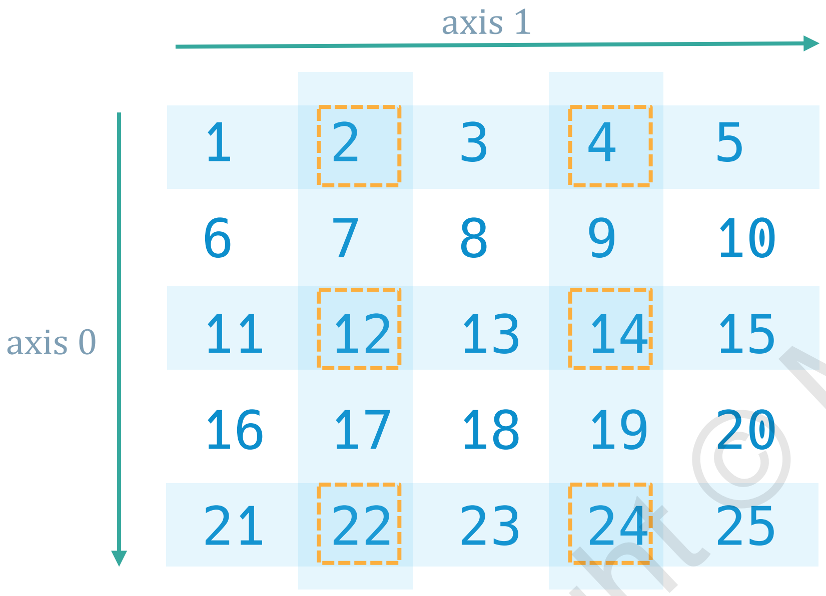
axis 1 slice

axis 0 slice

→ can also write this as `arr[:2, 1:]`

Slicing 2-D Arrays

→ can get even fancier when using steps



→ can think of this as the intersection of

→ rows 0, 2, 4 → `[::2]`

→ columns 1, 3 → `[1::2]`

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

Slice Assignment in NumPy Arrays

→ works very similarly to assigning to `list` slices

→ cannot replace with an array that is not the same shape

→ also means we cannot change size of the original array

→ makes sense since NumPy arrays are fixed size

→ be careful with data types!

```
a = np.array([1, 2, 3, 4, 5])
a[0:3] = np.array([10, 20, 30])  a → [10, 20, 30, 4, 5]
```

→ can also replace with a `list` or `tuple` – NumPy will handle it

Slice Assignment in NumPy Arrays

→ can also assign a single value (not an array) to a slice

→ NumPy basically fills the slice with the same value repeated as many times as necessary (this is called broadcasting)

```
arr = np.array([1, 2, 3, 4, 5, 6, 7])

arr[::3]   → [1, 4, 7]

arr[::3] = 0    arr → [0, 2, 3, 0, 5, 6, 0]
```

Slices are "linked" to Original Array

→ similar to `reshape` we saw earlier

→ a slice is "linked" to the array it was sliced from

```
arr = np.array([1, 2, 3, 4, 5])

s = arr[0:3] → [1, 2, 3]
```

→ replacing an element in `s` will be "seen" by `arr`

→ and vice versa

→ to avoid this, make a copy of the slice

```
s = arr[0:3].copy()
```

Coding

Fancy Indexing

→ we saw how to use single index values to specify an array item

1-D → `arr[3]`

2-D → `arr[2, 5]`

→ we saw how to use slicing

1-D → `arr[1:3:2]`

2-D → `arr[1:3:2, :5]`

→ single items at a time

→ items that can be defined using slicing

→ sometimes not enough – what if we want items (or rows) 1, 2 and 4?

One way…

```
arr = np.array([1, 2, 3, 4, 5, 6])
```

→ want an array consisting of elements at indices `0`, `1`, `3` and `5`

```
sub = np.array([arr[0], arr[1], arr[3], arr[5]])
```

→ works

→ but what we really have is an array of indices `np.array([0, 1, 3, 5])`

→ and NumPy supports specifying elements using an array of indices instead of just a single index

Fancy Indexing

→ use an array of indices (an index array)

```
arr = np.array([1, 2, 3, 4, 5, 6])

index_array = np.array([0, 1, 3, 5])

sub = arr[index_array]
```

→ 1, 2, 4, 6

→ can also just define the index array inline

```
sub = arr[np.array([0, 1, 3, 5])]
```

Array Index Shape

→ shape of array index determines shape of selection

```
arr = np.array([1, 2, 3, 4, 5, 6])

arr[np.array([0, 1, 3, 4])] → [1, 2, 4, 5]

                         (4, )

arr[np.array(
  [
    [0, 1],                        [
    [3, 4]          →                [1, 2],
  ]                                  [4, 5]
)                                  ]

                 (2, 2)
```

Fancy Indexing in Multiple Dimensions

→ fancy indexing can be applied to multiple axes

```
[index_array, index]          [index, index_array]

[index_array, slice]          [slice, index_array]

[index_array, index_array]
```

`index_array` **and** `index`

```
1  2  3  4  5
6  7  8  9  10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
```

`arr[1, np.array([0, 1, 3])]`

single row

→ `[6, 7, 9]`

```
1  2  3  4  5
6  7  8  9  10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
```

`arr[np.array([0, 1, 3]), 1]`

single column

→ `[2, 7, 17]`

→ note how resulting array is 1-D

`index_array` **and** `slice`

1	2	3	4	5
$\boxed{6}$	$\boxed{7}$	8	$\boxed{9}$	10
$\boxed{11}$	$\boxed{12}$	13	$\boxed{14}$	15
16	17	18	19	20
21	22	23	24	25

```
arr[1:3, np.array([0, 1, 3])
```

multiple rows

multiple columns

```
→ [
     [6,  7,  9],
     [11, 12, 14]
  ]
```

$\boxed{1}$	2	3	$\boxed{4}$	5
$\boxed{6}$	7	8	$\boxed{9}$	10
$\boxed{11}$	12	13	$\boxed{14}$	15
$\boxed{16}$	17	18	$\boxed{19}$	20
$\boxed{21}$	22	23	$\boxed{24}$	25

```
arr[:, np.array([0, 3])
```

```
→ [ [1, 4],
    [6, 9],
    [11, 14],
    [16, 19],
    [21, 24]
  ]
```

`index_array` and `index_array`

→ not commonly used – can be confusing for someone reading your code

→ keep index arrays same shape

1-D and 1-D

```
arr[np.array([0, 2]), np.array([1, 3])]
```

→ think of this as zipping the indices from the two axes

→ `(0, 1)  (2, 3)`

```
1  2  3  4  5
6  7  8  9  10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
```

→ `[2, 14]`

`index_array` and `index_array`

2-D and 2-D

→ again think of this as zipping up indices from both axes

→ but now our "index array" is really 2-D as well

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

```
0 1    0 2        (0, 0) (1, 2)
3 4    1 3   →    (3, 1) (4, 3)
```

```
 1   2   3   4   5       →   [
 6   7   8   9  10               [1, 8],
11  12  13  14  15               [17, 24]
16  17  18  19  20           ]
21  22  23  24  25
```

Coding

Masking

Boolean Masking

→ use an expression that evaluates to a boolean for each element of an array

→ make an array of those `True`/`False` values

→ use that array to "filter" elements in another array

	> 0	apply filter
10	True	10
-10	False	
20	True	20
-20	False	
30	True	30
-30	False	

Comparison Functions

→ functions which can be applied to each element of an array

→ returns an array containing the result for each element

```
np.less(arr, value)
```

→ looks at every element of `arr` and evaluates `element < value`

```
arr = np.array([1, 2, 3, 4, 5])

np.less(arr, 4)    → [True, True, True, False, False]
```

NumPy Logic Functions

→ other functions exist:

`greater`   `less_equal`   `equal`   `not_equal`

and more…

→ https://numpy.org/doc/stable/reference/routines.logic.html

→ but we can just use comparison operator symbols

`<`   `<=`   `>`   `>=`   `==`   `!=`

→ using these will use the NumPy corresponding functions

Applying the Mask

→ this array of `True`/`False` values is called a mask

→ we can apply this mask to an array (use same shaped arrays)

```
arr = np.array([1, 2, 3, 4])

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

→ or just use `mask = arr != 3`

`arr[mask]` → `[1, 2, 4]`

→ can do all this in a single statement `arr[arr != 3]`

Masking 2-D Arrays

→ masks will return a 1-D array, even if array being masked is 2-D

→ basically applies mask element by element

```
arr = [
  [1, 2],
  [3, 4]
]
```

```
mask = arr != 3
```

```
mask → [
        [True, True],
        [False, True]
       ]
```

```
arr[arr != 3]
```

→ [1, 2, 4]

→ result is 1-D

Combining Logical Operators

→ Python uses `and` `or` `not`

→ for NumPy we have to use

`&`	and
`|`	or
`~`	not (complement)

→ because of operator precedence, use `( )` to group logic expressions

```
arr = np.arange(-10, 10)

arr[(arr > 0) & (arr % 2 == 0)]
```

→ `[2, 4, 6, 8]`

Coding

Universal Functions

→ earlier we saw that universal functions are vectorized functions

- → they apply a function to each element of an array
- → the loop and function evaluation are done in C, not Python
- → very fast
    - → we'll see how much faster in coding section

- → NumPy has a large number of universal functions
    - → math operations (arithmetic, logs, exponentials, sqrt, abs, ...)
    - → trig and hyperbolic
    - → comparison functions (equal, less than, greater than, min/max, ...)

https://numpy.org/doc/stable/reference/ufuncs.html#available-ufuncs

Universal Functions and Operators

→ `add`, `subtract`, `multiply`, `divide`, `floor_divide`, `mod`, `power`, …

→ can be called as functions, with at least one argument being an array

```
np.add(arr_1, arr_2)
np.add(arr_1, scalar)
```

→ or just use the `+` operator

→ Python will use `np.add`

→ similarly with `-`, `*`, `/`, `//`, `%`, `**`

Array and Array

$[a_0, a_1, a_2] + [b_0, b_1, b_2] \rightarrow [a_0 + b_0, a_1 + b_1, a_2 + b_2]$

$[a_0, a_1, a_2] \% [b_0, b_1, b_2] \rightarrow [a_0 \% b_0, a_1 \% b_1, a_2 \% b_2]$

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix} ** \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \end{bmatrix} \rightarrow \begin{bmatrix} a_{00} ** b_{00} & a_{01} ** b_{01} & a_{02} ** b_{02} \\ a_{10} ** b_{10} & a_{11} ** b_{11} & a_{12} ** b_{12} \end{bmatrix}$$

→ keep array shapes the same

→ technically possible to use different shapes → broadcasting

https://numpy.org/doc/stable/user/basics.broadcasting.html

Array and Scalar

→ simplest form of broadcasting

```
[a1, a2, a3] * 3   →  [a1, a2, a3] * [3, 3, 3]
```

scalar

broadcast to match shape

$$1 \;/\; \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \;/\; \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix}$$

Mismatched Shapes

→ sometimes possible → not going to focus on this in this course

same number of elements

```
a1 = [
      [1, 2, 3],
      [4, 5, 6],
      [7, 8, 9]
     ]
```

```
a2 = [10, 20, 30]
```

```
                 [                        [
                  [1, 2, 3],               [10, 20, 30],
a1 * a2    →      [4, 5, 6],      *        [10, 20, 30],
                  [7, 8, 9]                [10, 20, 30]
                 ]                        ]
```

Coding

Additional Math and Stats Functions

→ NumPy has a host of array manipulation and computational functions

- → trig/hyperbolic, logs/exponents
- → linear algebra (matrix/vector products, eigenfunctions/values, inverses, etc)
- → stats (averages, variances, correlations, histograms)
- → discrete Fourier transforms

  https://numpy.org/doc/stable/reference/routines.html

- → <u>simple</u> financial functions
  - → mainly related to interest calculations
  - → slated to be removed from NumPy → don't use them

Other More Specialized Libraries

→ many more specialized libraries

→ usually built on top of NumPy and Pandas

→ SciPy interpolation, optimization, integration, linear algebra, stats, …

→ statsmodels regression, imputation, models, time series, …

→ pyfolio portfolio performance and risk analysis

→ QuantLib quantitative financial library

→ Quandl useful for getting financial datasets directly into Python (not all datasets are free)

Axes

→ recall discussion on axes

```
l = [
      [1, 0, 0],
      [0, 1, 0],
      [0, 0, 1]
    ]
```

rows

columns

→ also called axes

rows → axis 0

columns → axis 1

→ many of the universal functions in NumPy can operate

→ on the array as a whole

→ along an axis

Max

→ 1-D is intuitive `np.amax(np.array([1, 2, 3]))` → `3`

`arr` →

$$\begin{array}{cc} & \text{axis 1} \\ \text{axis 0} & \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \end{array}$$

`np.amax(arr)` → `12`

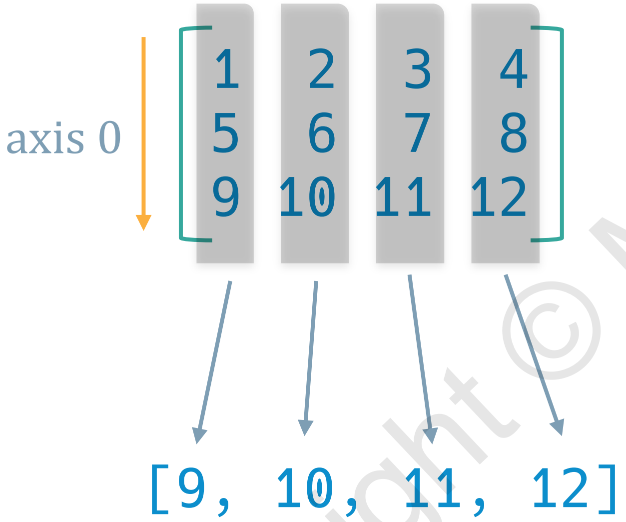
→ simply runs through all elements of array

Max

→ can specify an axis

`np.amax(arr, axis=0)` → performs the operation across each row

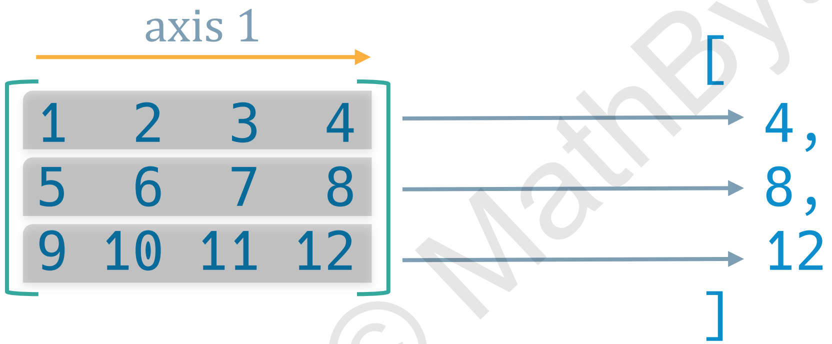
(i.e. for each column)



Max

`np.amax(arr, axis=1)` → performs the operation across each column

(i.e. for each row)



Other Functions

→ some functions only operate element by element

`sin` `sinh` `arcsin` `arcsinh`

`log` `exp` `around` …

→ some functions, like `amax`, that operate on groups of data, support axes

`amax` `amin`

`mean` `median` `std`

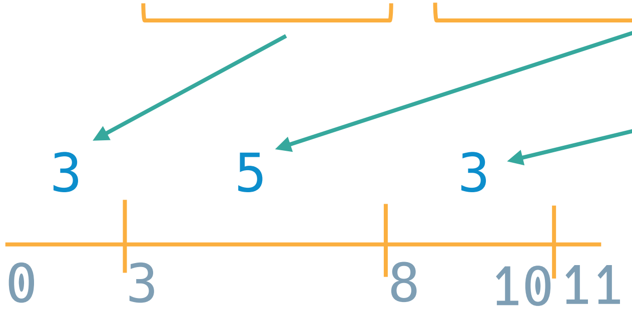
`sum` `cumsum` `product` …

https://numpy.org/doc/stable/reference/routines.math.html

Histogram

→ `np.histogram` → creates binned frequency distribution

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```



bins → `[0, 3)  [3, 8)  [8, 11)`

→ define bin bounds using left edge, and rightmost edge (which is inclusive)

→ `bins = [0, 3, 8, 10]`

`np.histogram(a, bins_arr)` → tuple: (array frequencies, bins array)

`np.histogram(a, int)` → calculates evenly spaced bins in min/max range

→ tuple: (array frequencies, bins array)

→ other variants https://numpy.org/doc/stable/reference/generated/numpy.histogram.html

Coding

Pandas

28

→ Pandas is built on top of NumPy

→ data manipulation and analysis, focused on tabular and time series data

→ arrays with rows and columns

→ but uses labels to identify rows and columns

→ in addition to positional indices

→ columns in the same array can have different data types

`Series` → 1-dimensional

`DataFrame` → 2-dimensional

→ a collection of `Series` objects

`Index` → used to index `Series` and `DataFrame` objects

one of the key differences between Pandas and NumPy

→ NumPy array elements are indexed (implicitly) by position

→ in Pandas we can assign our own (explicit) labels

→ this section will cover some of the basics of Pandas

→ Pandas is a huge library

→ lots of data querying and manipulation functionality

https://pandas.pydata.org/

→ user guide https://pandas.pydata.org/docs/user_guide/index.html

→ API reference https://pandas.pydata.org/docs/reference/index.html

Indexes

→ let me get something out of the way first ☺

→ index

→ indexes? → indices?

→ both are correct

→ I am not always consistent!

usually…

→ I refer to elements of an index as indices

→ I refer to multiple index objects as indexes

What is an index?

→ arrays / lists

```
['a', 'b', 'c', 'd']
  0    1    2    3
```

element in array can be identified by its (positional) index

→ dictionaries

value in a dictionary can be identified via its key

```
{
  'a': 1,
  'b': 2,
  'c': 3
}
```

→ an index is a way to "look up" one or more values in an array or dictionary

Sequence Types

→ sequence types such as Python lists, tuple and NumPy arrays

→ have a natural positional order to their elements

→ this forms an implicit index on the sequence

```
l = ['a', 'b', 'c', 'd']
      0    1    2    3
```

```
l[0]
l[1:4]
```

uses the positional indices

→ with Pandas we can define an explicit index (in addition to the implicit index)

```
idx = ['first', 'second', 'third', 'fourth']
          ↕          ↕         ↕         ↕
  l = [  'a',       'b',      'c',      'd'  ]
```

→ we'll see how this works later

Pandas Indexes

→ `pd.Index` → most generic type of `Index`

→ they contain elements

→ they are based on NumPy arrays

→ they themselves have an implicit positional index

```
idx = pd.Index([10, 20, 30, 40])
```

Python list. tuple, NumPy array, ...

```
idx[0] → 10
idx[1:4] → Index([20, 30])
idx[[0, 2]] → Index([10, 30])
idx[idx % 4 == 0] → Index([20, 40])
```

returns an `Index` object

Specialized Indexes

→ Int64 indexes for indexes that contain integer indices

→ Float64 indexes for indexes that contain float indices

→ Range indexes for integer sequence defined via a range

→ similar to difference between Python `list` and `range`

`[0, 1, 2, 3, 4, 5]` → sequence is materialized

`range(0, 6)` → sequence is not materialized

→ elements are produced as requested when iterating

→ Range indexes can be more efficient (storage and computation)

Indexes Have Set-Like Properties

→ can find the union and intersection of indexes

`&` → intersection

`|` → union

`in` → element of

→ Pandas will use broadest data type needed for union/intersection

→ `RangeIndex` indexes will try to return a `RangeIndex` as result of union/intersection

→ not always possible

String, Integer and Float Indexes

→ strings will result in an `Index` object, with an `object` data type (a catchall type)

```
pd.Index(['a', 'b', 'c'])
```

→ integers will result in an `Int64Index` object

```
pd.Index([1, 2, 3])
```

→ floats will result in a `Float64Index` object

```
pd.Index([0.1, 0.2, 0.3])
```

Range Indexes

→ can create using the Python `range` object

```
pd.Index(range(1, 10, 2))
```

→ can use Pandas `RangeIndex` class directly

```
pd.RangeIndex(start, stop, step)
```

→ index values do not have to be unique

`pd.Index([1, 1, 2, 2])` → perfectly legal

but if we associate an index with a sequence, how does a non-unique index work?

A ⟷ 10
B ⟷ 20
A ⟷ 30
B ⟷ 40

item at index A? → two items! → [10, 30]

→ an index value may refer to multiple values in the associated array

→ a bit different from Python dictionaries

Coding

Series

Python Sequences, NumPy Arrays

→ associative arrays

```
l = [10, 20, 30, 40, 50]
      0   1   2   3   4

a = np.array([10, 20, 30, 40, 50])
               0   1   2   3   4
```

those sequences can be used to access (or reference) items in the sequence

→ they are also called indexes

→ based on position

→ positional index

→ there is an association between the index and the values → associative array

→ in Python lists, tuples, NumPy arrays, this positional index is implicit

→ index provides a unique mapping between indices and values

Python Dictionaries

→ another type of associative array

→ mapping between keys and values

→ keys are not positional based

→ do not even have to be numbers

→ but it's still an associative array

```
d = {'a': 1, 'b': 2, 'c': 3}
```

a → 1
b → 2
c → 3

index

→ unique index

→ no implicit positional index

Pandas Series

→ another type of associative array

  → has some dictionary-like properties

  → has some sequence-like properties

→ it's a sequence type – so elements have a definite position in collection

  → positional index

→ can also define an explicit index → a second index

0	10	a
1	20	b
2	30	c
3	40	d

implicit positional index

→ it's always there

explicit custom index

→ indices are also referred to as labels

```
0    10    a
1    20    b
2    30    c
3    40    d
```

→ can reference items by positional indices `[0]` `[1]` …

→ or by using the explicit index `['a']` `['b']` …

→ can even use slicing and fancy indexing

→ even with an explicit index that is not numerical

```
['a':'c']

['a':'d':2]
```

→ indexing works as expected

→ slicing has a twist

→ positional index `[0:5]` → excludes endpoint

→ explicit index `['a':'c']` → includes endpoint

0	1	2	3
10	20	30	40
a	b	c	d

`[0:2]` → `10, 20`

`['a':'c']` → `10, 20, 30`

Pandas understands that these are not positional indices (strings)

A point of confusion…

```
  0    1    2    3
[100, 200, 300, 400]
  2    3    4    5
```

implicit index

explicit index

`[2]` → is this using implicit index?

`[2:3]` → or explicit index?

if both implicit and explicit index are integers:

`[2]` → uses explicit index

`[2:3]` → uses implicit index

→ can be confusing

`loc` and `iloc` attributes

→ allows us to specifically indicate use of implicit or explicit index

```
        0     1     2     3
s =   100,  200,  300,  400
        2     3     4     5
```

implicit index

explicit index

`s.iloc[2]` → uses implicit index

`s.loc[2]` → uses explicit index

note the square brackets

Deleting Items

→ indexes are immutable

→ deleting an item would require deleting the corresponding index value

→ instead use `.drop( )` method

→ returns a new series with new explicit index

Creating Series objects

```
from pandas import Series
```

→ from a dictionary

```
Series({'a': 1, 'b': 2})
```

```
0 1
1 2
a b
```

implicit index

explicit index

→ from a list, specifying explicit index using another list

```
Series([1, 2], index=['a', 'b'])
```

Series Attributes and Methods

`.index` → returns the explicit Index object

`.values` → returns a NumPy array of the values

`.items` → zip of explicit index values and array values

`.iloc` → used for indexing using implicit index

`.loc` → used for indexing using explicit index

`.drop` → used to remove an element by explicit index

Coding

DataFrames

→ `Series` → analogous to 1-D NumPy array with an explicit index

→ `DataFrame` → analogous to a 2-D NumPy array with an explicit index

→ for the rows

→ and for the columns

another way to look ay it…

a `DataFrame` is a collection of `Series` objects

→ a common explicit index for the rows → series are aligned

→ the columns (Series) form a Series too → explicit index

→ column names possibly

explicit index for rows

explicit index for columns

implicit index for rows

	county	population	gdp	area	
The Bronx	Bronx	1,418,207	42.695	42.10	0
Brooklyn	Kings	2,559,903	91.559	70.82	1
Manhattan	New York	1,628,706	600.244	22.83	2
Queens	Queens	2,253,858	93.310	108.53	3
Staten Island	Richmond	476,143	14.514	58.37	4
	0	1	2	3	

implicit index for columns

→ can think of it as a Series of Series

→ or a dictionary of dictionaries

column index labels

```
{
  'county': {
              'The Bronx': 'Bronx',
              'Brooklyn': 'Kings',
               …
             },
  'population': {
                  'The Bronx': 1_418_207,
                  'Brooklyn': 2_559_903,
                   …
                 },
  'gpd': { … },
    …
}
```

row index labels

Constructing a DataFrame

`pd.DataFrame(…)`

→ from a list of Series objects

→ from a list of lists

→ from a list of dictionaries

→ from a dictionary of Series objects

→ from a dictionary of dictionaries

  → in some cases row and column explicit indexes are created as expected

  → in some cases we may have to define these indexes manually

Some `DataFrame` Properties and Methods

`.info()` → prints some useful info about the data frame

`.transpose()` → transposes the data frame, maintaining indexes

`.rename()` → allows us to rename the index labels (rows and/or columns)

`.set_index()` → use an existing column in the data frame as a row index

`.index` → the Index object used to index the rows

`.columns` → the Index object used to index the columns

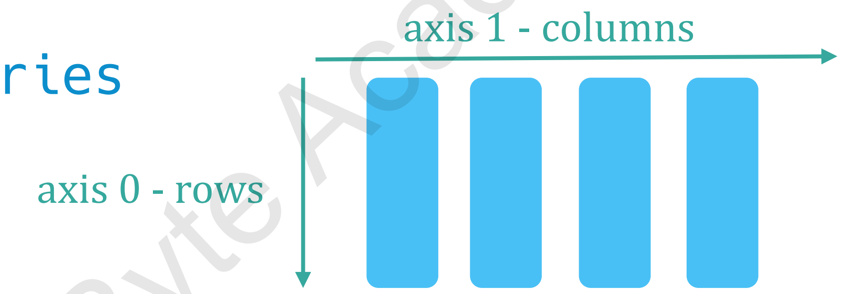
`.drop()` → used to drop rows/columns from the data frame

Coding

Selecting Data

DataFrames

→ analogous to a `Series` of `Series`



→ or a dictionary of lists / dictionaries

```
{
  "col0": {
            "row0": value,
            "row1": value
          },
  "col1": {
            "row0": value,
            "row1": value
          }
}
```

→ a sequence of aligned columns

→ consider it as a `Series` of `Series`

$$\texttt{df} = \begin{array}{cccc} & \texttt{c1} & \texttt{c2} & \texttt{c3} \\ \texttt{r1} & 1 & 2 & 3 \\ \texttt{r2} & 4 & 5 & 6 \\ \texttt{r3} & 7 & 8 & 9 \end{array}$$

→ `c1` is a series of values
→ `c2` is a series of values
→ `c3` is a series of values

share a common row index `['r1', 'r2', 'r3']`

→ `df` is like a Series `[c1, c2, c3]` with index `['c1', 'c2', 'c3']`

→ or like a dictionary `{'c1': c1, 'c2': c2, 'c3': c3}`

`df['c1']` → this selects the item with label `'c1'`

→ the column (series) `c1`

→ note that `[]` cannot be used with positional indices with `DataFrame` objects

`loc` and `iloc`

→ just like with `Series`, but with 2 axes

→ `loc` uses the explicit index

→ `iloc` uses the implicit (positional) index

→ but think of `DataFrame` like a NumPy array with two axes

axis `1` (columns)

`df.loc[v1, v2]`   `df.iloc[v1, v2]`

axis `0` (rows)

→ slicing and fancy indexing works the same way as with `Series`, but using 2 axes

Replacing Values

→ can replace values using assignment (`==`) operator

→ replace single selected cell

  → with a scalar value

→ replace multiple cells selected using slicing/fancy indexing

  → with a 2-D NumPy array/list of lists of same shape

  → with a scalar value that will be broadcast

  → with a 1-D NumPy array that will be broadcast

  → can replace with a `Series` or `DataFrame` but indexes can cause issues!

Coding

Missing Values

Python

→ `None` object → can be used to indicate undefined or missing in a sequence

`[1, 2, None, 4]`

→ IEEE standard for floats also has the concept of an undefined float

→ NaN (not a number)

→ `float('nan')`

→ `math.nan`

→ `np.nan`

Equality of NaN

→ two NaN values always compare `False`

→ cannot compare two undefined (unknown) values…

```
a = math.nan        a == b → False
b = math.nan        a is b → False
```

→ so how do we test if a number is NaN?

→ `math.isnan( )`

`math.isnan(np.nan)` → `True`

→ NumPy universal function `np.isnan( )`

Pandas Series

→ if the series is a series of floats

`nan` → `nan`
`None` → `nan`

series was made into a float

```
pd.Series([1, 2, None, np.nan])
```

→ `[1.0, 2.0, NaN, NaN], dtype=float64`

→ if the series is a series of `object` (for example for series of strings)

```
pd.Series(['a', 'b', None, np.nan])
```

`None` was not converted to `NaN`

→ `['a', 'b', None, NaN], dtype=object`

Testing for Missing Data

→ could be `None` → could be `NaN`

→ `pd.isnull()` → handles both

→ universal function (operates on `Series` or `DataFrames`)

→ returns element by element comparison

→ `True` if value is `None` or `NaN`

→ `pd.notnull()` → similar to `isnull()`, but opposite result

Replacing `Series` Missing Data

→ use loops to iterate and replace missing values

→ specialized Pandas functions

→ `s.fillna(value)` → replaces any null with specified `value`

→ `s.fillna(method=…)`

→ `method = 'ffill'`

→ forward fill `null, 1, null, 2, null, null`

→ `[null, 1, 1, 2, 2, 2]`

→ `method = 'bfill'` → backward fill

Replacing `DataFrame` Missing Data

→ works same as `Series` replacement

→ but the `axis` is important for back/forward fills

back/forward fill along axis 1

or along axis 0

0.0	0.1	0.2	0.3
1.0	NaN	1.2	1.3
2.0	2.1	NaN	2.3
3.0	3.1	3.2	3.3

```
df.fillna(method='ffill', axis=0)    2.0 2.1 1.2 2.3

df.fillna(method='ffill', axis=1)    2.0 2.1 2.1 2.3
```

Interpolating Missing Data

→ more advanced techniques

→ linear interpolation

→ splines ...

→ beyond scope of this course

→ but will look at simple linear interpolation in code

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.interpolate.html

Dropping Data

→ already saw this for `Series` objects

→ `DataFrame` is 2-D

0.0	0.1	0.2	0.3
1.0	NaN	1.2	1.3
2.0	2.1	NaN	2.3
3.0	3.1	3.2	3.3

→ do we delete rows with missing values?

→ or do we delete columns with missing values?

→ need to specify an axis

```
df.dropna(axis=0)
df.dropna(axis=1)
```

→ axis defaults to `0` if we don't specify it

Coding

Loading Data

→ Pandas has built-in functions for loading many types of data

in this lecture we'll look at

→ CSV files

→ Excel files

→ many other data sources are supported (SQL, JSON, SAS, SPSS, etc)

https://pandas.pydata.org/pandas-docs/stable/reference/io.html

Loading a CSV File

→ `pd.read_csv(<file_name>)`

→ has many optional arguments

- → `sep` and `delimiter` (just like Python's `csv.reader`)
- → `header` row number to use as column labels, otherwise infers them
- → `usecols` a list of positional indexes indicating which columns to keep
- → `names` renames the columns
- → `index_col` specifies (by name or index) which column to use as the row index

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

Loading an Excel File

→ Pandas relies on external 3rd party libraries to read Excel files

→ many exist, such as `xlrd`, `openpyxl`

→ need to `pip install` the library in your virtual env

→ already done if you followed install at beginning of course

→ `pd.read_excel('file_name')`

→ `sheet_name` the sheet name, or index (zero based) to load

→ `header`

→ `usecols`

→ `names`

→ `index_col` and more…

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_excel.html

Coding

Basic Data Analysis

→ basic facts about a loaded data set

`.info()` →column names, types, not-null counts

`.describe()` → mean, min, max, quartiles, std dev

→ by default only includes numerical columns

→ `include='all'`

→ categorical columns

→ # unique values

→ most frequent value + frequency

→output is "print" output

→ equivalent methods to obtain the same data

`.nunique( )` → # of unique values

`.unique( )` → array of unique values

`.value_counts( )` → Series of values and their frequency

`.count( )`

`.mean( )`

`.std( )`

`.quantile( )`

Coding

Sorting and Filtering

Filtering

→ boolean masking

→ works similarly to NumPy and Series masking

→ create a boolean masking array

→ apply mask to data frame

→ use explicit or implicit index

```
mask = df['col'] >= 0

mask = df.iloc[:, 2] >= 0

df[mask]
```

Sorting

→ sort rows based on the row index labels

```
df.sort_index( )
```

→ sort rows based on values in a column

```
df.sort_values('col label')
```

→ similarly to Python's `sorted( )` function, these support a `key` argument

Reviewing `sorted(key=…)`

```
l = ['Z', 'a', 'b']

sorted(l, key=lambda x: x.casefold())
```

→ `key` is a function that transforms each element of `l`, one by one

```
        l = ['Z', 'a', 'b']

sort_keys = ['z', 'a', 'b']
```

→ sorting is then based on `sort_keys`

→ sort by an associated series of keys

The key Argument for DataFrames

→ sort by an associated series of keys

→ instead of using a function that generates the keys one by one

→ use a vectorized function to generate the sequence of sort keys all at once

→ key function receives a `Series` as its argument

→ should return a `Series` object with same shape

```
s = Series([1, -1, 2, -2])

key = np.abs(s)    → Series([1, 1, 2, 2])
```

Sorting by Index

```
a  1  2  3
B  4  5  6
c  7  8  9
```

```
def sort_func(ind):
  return ind.str.casefold()
```

```
sort_func(df.index)
   → ['a', 'b', 'c']
```

```
df.sort_index(key=sort_func)
```

or

```
df.sort_index(key=lambda ind: ind.str.casefold())
```

Sorting By Values

→ same as sorting by index

→ uses some specified column instead of index

	c1	c2	c3
a	1	-2	3
B	-4	5	6
c	7	8	-9

```
df.sort_values('c1')
```

→ sorts based on values in `c1`

	c1	c2	c3
B	-4	5	6
a	1	-2	3
c	7	8	-9

index is preserved

Sorting by Values with a key

→ key function receives the sort by column (Series) as its argument

```
   c1    c2   c3
a  1    -2    3
B  -40   5    6
c  7     8   -9
```

```
df.sort_values('c1', key=lambda col: np.abs(col))
```

→ key function receives column c1 as its argument

→ returns a new Series → 1, 40, 7

```
    c1  c2   c3
a   1   -2    3
c   7    8   -9
B   -40  5    6
```

Sorting on Multiple Columns

→ can specify a multi-level sort based on multiple columns

c1	c2	c3
a	-1	100
z	2	200
a	-3	300
a	10	400
z	-1	500

```
df.sort_values('c1')
```
→ stable sort based on c1 column

c1	c2	c3
a	-1	100
a	-3	300
a	10	400
z	2	200
z	-1	500

```
df.sort_values(['c1', 'c2'])
```
→ sorts on c1, then c2

c1	c2	c3
a	-3	300
a	-1	100
a	10	400
z	-1	500
z	2	200

Coding

Manipulating Data

→ vectorized operations similar to NumPy arrays

`.count` `.sum` `.prod`

`.min` `.max` `.mean` `.std`

→ works across all elements of `DataFrame`

→ or along a specified `axis`

→ regular arithmetic operators

→ NumPy universal functions

→ `.transpose()`

```
   a b c               A B C
A  0 1 2        →   a  0 3 6
B  3 4 5            b  1 4 7
C  6 7 8            c  2 5 8
```

→ `to_numeric()`

→ int, float

→ entries that cannot be converted result in an exception

→ can override this behavior using the `errors` argument

`errors = 'coerce'`

Concatenating DataFrames

→ concatenate along an axis

```
pd.concat([df1, df2, …], axis=0|1)
```

`axis = 1` → horizontally   `axis = 0` → vertically

→ uses row or column index to "align" concatenated rows/columns

axis = 1 →

	a	b	c
r1	1	2	3
r2	4	5	6
r3	7	8	9

	d	e	f
r1	1	2	3
r2	4	5	6
r4	7	8	9

→

	a	b	c	d	e	f
r1	1	2	3	1	2	3
r2	4	5	6	4	5	6
r3	7	8	9	nan	nan	nan
r4	nan	nan	nan	7	8	9

→ this is called an outer join

Concatenating DataFrames

→ in an outer join "missing" data in the join are replaced with `NaN`

→ outer joins are the default

	a	b	c
r1	1	2	3
r2	4	5	6
r3	7	8	9

	d	e	f
r1	1	2	3
r2	4	5	6
r4	7	8	9

	a	b	c	d	e	f
r1	1	2	3	1	2	3
r2	4	5	6	4	5	6
r3	7	8	9	nan	nan	nan
r4	nan	nan	nan	7	8	9

→ in an inner join, missing rows/columns are dropped entirely

	a	b	c
r1	1	2	3
r2	4	5	6
r3	7	8	9

	d	e	f
r1	1	2	3
r2	4	5	6
r4	7	8	9

	a	b	c	d	e	f
r1	1	2	3	1	2	3
r2	4	5	6	4	5	6

```
pd.concat([df1, df2], axis=1, join='inner')
```

Coding

Matplotlib

29

→ `Matplotlib` is a popular graphing library

https://matplotlib.org/

→ integrates well with Jupyter Notebooks

→ there are many others available too

→ geoplotlib	maps, geographical data
→ ggplot	little simpler than matplotlib, not as customizable (based on matplotlib)
→ plotly	interactive plots/web, contour plots, 3D, …

and more…

→ numerous extension packages to Matplotlib

- → financial
- → maps and map projections
- → specialty axes (like broken axes)
- → electronic circuits
- → Venn diagrams
- → density maps
- → statistical maps
- → ML visualizations

and many more…

https://matplotlib.org/3.1.0/thirdpartypackages/index.html

→ we'll look at how to create and theme various Matplotlib charts

→ single plots

→ overlayed plots

→ grids of plots

→ we'll look at OHLC plots using `mplfinance` extension

https://github.com/matplotlib/mplfinance

→
```
pip install matplotlib
pip install mplfinance
```

Matplotlib Basics

Imports

→ two sections of the `matplotlib` library we will use often

```
import matplotlib as mpl

import matplotlib.pyplot as plt
```

Styles

`mpl.style.available`

→ returns a list of the various styles available on your system

→ a list of strings

use the exact string as listed above

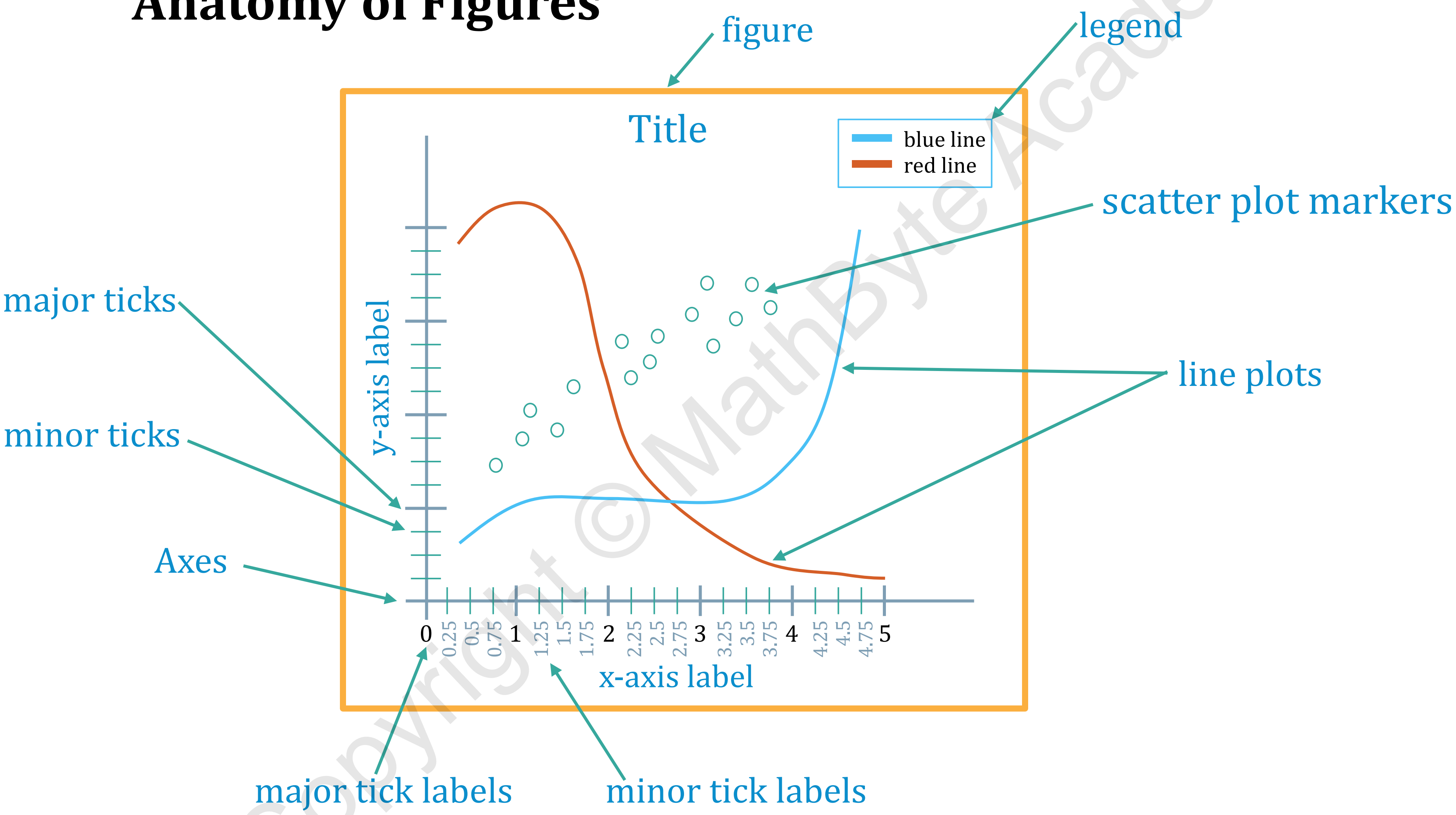
`mpl.style.use('…')`

→ sets your notebook to use a particular style

→ see a preview of various styles

https://matplotlib.org/3.2.1/gallery/style_sheets/style_sheets_reference.html

Anatomy of Figures



Creating a Figure and Axes

→ simplest is to use `subplots()` function in `pyplot` module

```
import matplotlib.pyplot as plt

plt.subplots()
```

→ creates a new figure and one `Axes` object

→ returns it as a `tuple`

→ and displays the figure in Jupyter

```
fig, ax = plt.subplots()
```

→ blank chart

→ need to specify something to plot

Plotting Data

→ we add a plot to an `Axes`

```
ax.plot(x_coords, y_coords, label='…')
```

→ this adds a (line) plot to the `Axes` object, with specified plot name (used in legend)

→ x and y coordinates can be lists, NumPy arrays, Pandas columns, …

→ can keep adding more plots to same Axes

→ we have to display the `figure` to see the result

→ typically create figure and plots in a single Jupyter cell

Additional Axes Settings

`ax.set_xlabel('…')` → sets the x-axis label

`ax.set_ylabel('…')` → sets the y-axis label

`ax.set_title('…')` → sets the title

`ax.legend()` → creates and adds a legend

→ note how all these are applied to the `Axes` object

→ later we'll see how to add multiple `Axes` to the same `figure`

Coding

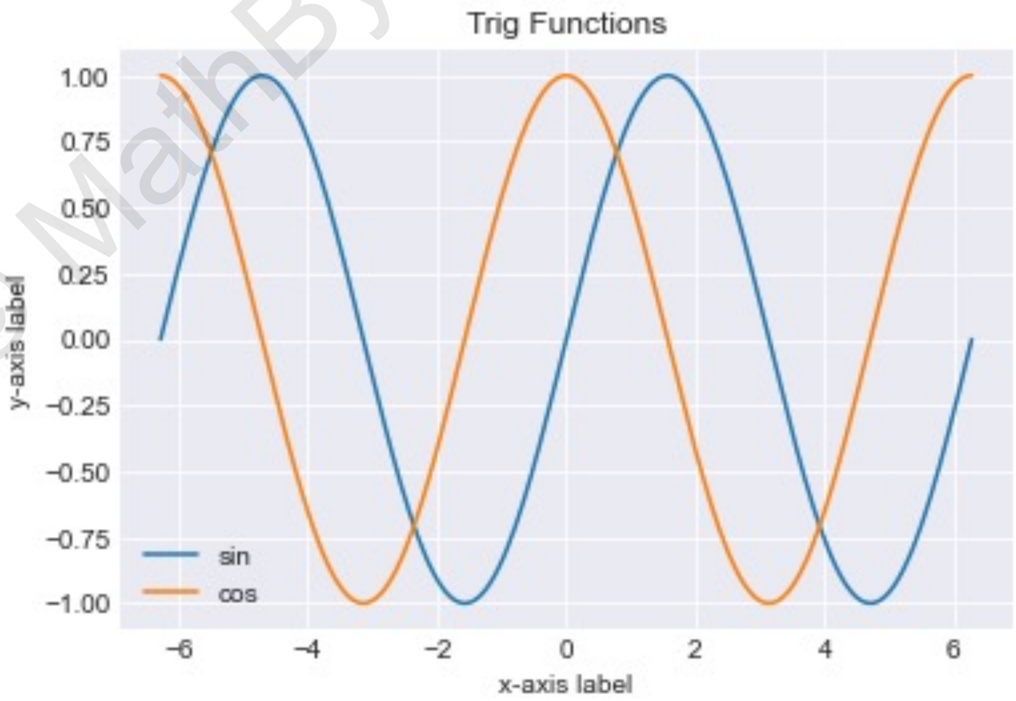
Multi Plots

Multiple Plots on Same Axes

→ saw this in previous lecture

→ just keep adding plot to same `Axes` object

```
ax.plot(…)
ax.plot(…)
```



Multiple Axes on Same Figure

→ we can also chart multiple `Axes` on the same figure

→ grid layout

→ number of columns → number of rows

```
plt.subplots(n_rows, m_columns)
```

→ creates the `figure`

→ creates `n * m` `Axes` objects laid out as specified

→ returns `figure` and collection of `Axes` as a 2-value tuple

→ first element is the `figure`

→ second element is a NumPy `ndarray` with all the `Axes`

Setting Figure Size

→ technically size is defined in width and height in inches

→ what that shows up as on your screen will depend on your resolution (dpi)

→ default is `6.4 (w) x 4.8 (h)`

→ can specify for a single figure

```
plt.subplots(figsize=(width, height))
```

→ or specified as a global change

```
plt.rcParams['figure.figsize'] = [width, height]
```

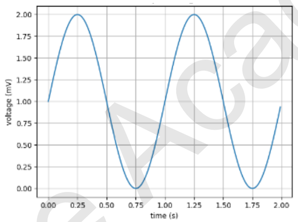
→ play around with width/height until you find a setting you like

Coding

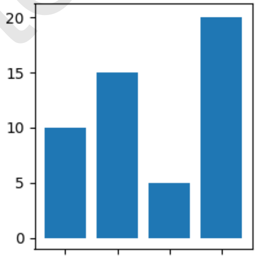
More Plot Types

Plot Styles

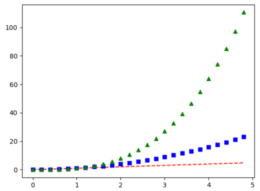
→ `plot(…)` a line plot



→ `bar(…)` vertical bar plot



→ `scatter(…)` scatter plot



→ plenty more…

https://matplotlib.org/3.1.1/gallery/index.html

Adding Vertical/Horizontal Lines to Axes

→ sometimes useful to add vertical/horizontal lines to a chart

→ display info such as mean, median, other important "values"

```
ax.axhline(y=…, xmin=0, xmax=1)
ax.axvline(x=…, ymin=0, ymax=1)
```

`xmin/xmax` → values between `0` and `1`

→ `0` indicates left edge, `1` indicates right edge

`ymin/ymax` → values between `0` and `1`

→ `0` indicates bottom edge, `1` indicates top edge

Histograms

→ can use NumPy to generate histogram data, and then use Matplotlib

→ but also built-in to Matplotlib directly

```
ax.hist(data, bins=…)
```

→ `data` is the data for which we want to generate the histogram

→ `bins` specifies the number of bins we want to use

Coding

Charting with `mplfinance`

`mplfinance`

https://github.com/matplotlib/mplfinance

→ add-on to Matplotlib that provides extra plot types

```
pip install mplfinance
```

→ import it in Jupyter

```
import mplfinance as mpf
```

→ we'll use it for candlestick/OHLC charts

Plotting OHLC Charts

→ simplest is to arrange a Pandas data frame as follows:

→ index is the datetime for each row

→ five data columns in specific order

→ Open, High, Low, Close, Volume

```
mpf.plot(data_frame)
```

Additional Plot Arguments

→ `type` used to specify chart type (e.g. `'ohlc'`, `'candle'`)

→ `mav` used to superimpose one or more moving averages
→ single value for single mav, tuple of values for multiple

→ `volume` `True` to display Volume bar chart (defaults to `False`)

→ `show_nontrading` `True` to show non-trading days in chart (gaps), defaults to `False`

Superimposing Plots

→ create subplots

→ `plots = mpf.make_addplot(…)`

→ add them to main plot when creating it

→ `mpf.plot(…, addplot=plots)`

Coding

Conclusion

30

Congratulations!!!

What we covered

→ the Python language

→ basic and advanced data types

→ some of the Python standard library

→ some popular 3rd party libraries

- requests → Web and API requests, JSON
- Numpy → efficient array computations
- Pandas → loading and manipulating data
- Matplotlib → charting data

Important References

www.python.org

numpy.org

pandas.pydata.org

matplotlib.org

requests.readthedocs.io/en/master/

Practice!

→ to learn programming: practice, practice, practice

- → yes, it can be hard
- → yes, you'll make mistakes
- → even seasoned devs struggle and make mistakes – often!

→read (and understand) other peoples' code

- → if you work with other devs, review each other's code
- → or find a developer friend who can do that
- → look at open source projects (GitHub)

→ be patient – don't give up

- → you won't become an expert in 3 weeks
- → as you write more code, you'll become more and more proficient

Practice Sites

→ there are many sites where you can practice with coding challenges

edabit.com

coderbyte.com

www.codewars.com

www.hackerrank.com

and many more…

Additional Resources

stackoverflow.com → if you have a coding question

→ you probably weren't the first with that question

→ you will probably find an answer, at least close

→ if not, you can post your question

→ just browse questions/answers – incredibly informative

Python Cookbook, by Beazley and Jones (O'Reilly Press)

Fluent Python, by Ramalho (O'Reilly Press)

YouTube → experts such as Hettinger, Beazley, Martelli, PyCon talks

Twitter → Raymond Hettinger (@raymondh)

Coding

almost kidding!

→ try executing these (separately) in a Jupyter cell

```
import this

import __hello__

import antigravity
```

→ if you're a dev who would rather use braces `{}` for code blocks

```
from __future__ import braces
```

I hope you enjoyed this course as much as I enjoyed creating it

Thank You!!