Intro To Python

Luca Coviello

WS19

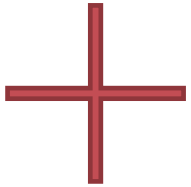
Roma, Oct 7 2019

@lucoviello

/in/luca-coviello

today

Practice!
Practice!
&
Practice!



- Python, but particularly, why Python?
- installing and first steps
- variables, data structures and constructs
- functions and object-oriented programming
- time to practice even more!

Special thanks to: Valerio Maggio (@lerioma) and Lorenzo Gaifas (@brisvag)

Python Programming Language

https://www.python.org

- High-level Programming Language
  - Very shallow learning curve
- Multi Platform
  - Windows, Linux, Unix (macOS)
- Off-side-rule based
  - i.e., no brackets nor semicolon required
- Language of the choice in:

Massachusetts Institute of Technology

NASA

Google

Named after the (Monty) Python

why Python?

- software quality
  - focus on readability
  - readability counts (from the Zen of Python)
  - minimalist programming approach
    - there's just one obvious way to do it
- program portability
  - (most) Python programs ruin unchanged on all mayor computer platforms
  - support for portable GUI, DB Access, web-based systems, ...
- developer productivity
  - 1/3 to 1/5 over Java, C++, C Code
- support libraries
  - great standard library
- component integration
  - glue language (e.g. data science)
  - can be easily integrated with C/C++ Code

still, why Python?

enjoyment | the act of programming is more pleasure than chore

is Python a Scripting Language?

- Python is a general purpose programming language
  - often applied in scripting roles
  - general purpose programming language that blends procedural functional and object-oriented paradigms

- but what do people think about it?
  - shell tools
  - control language
  - ease of use

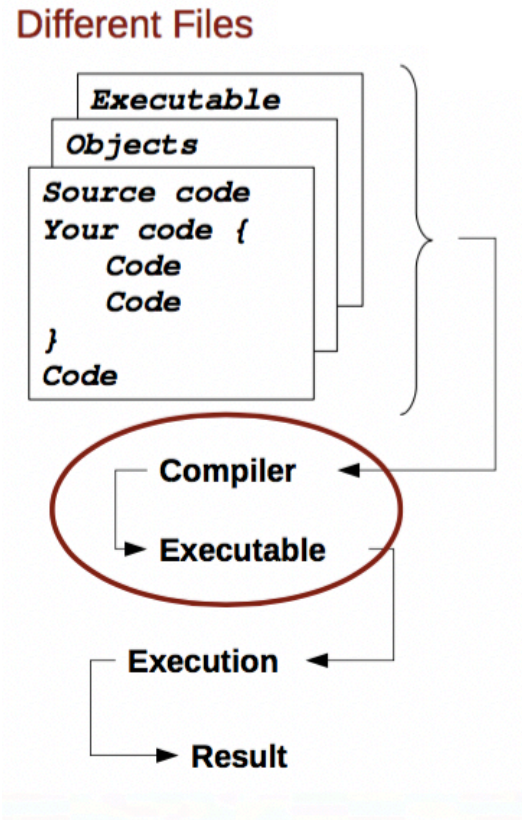
is Python a Scripting Language?

- shell tool
  - code operating-system-oriented scripts
  - it can! just like dozens of other application domains
- control language
  - “glue layer” to control other application components
  - it can! but it isn’t a control language
- ease of use
  - simple language for quick coding tasks
  - it is! but it isn’t only for simple tasks!

Python main features

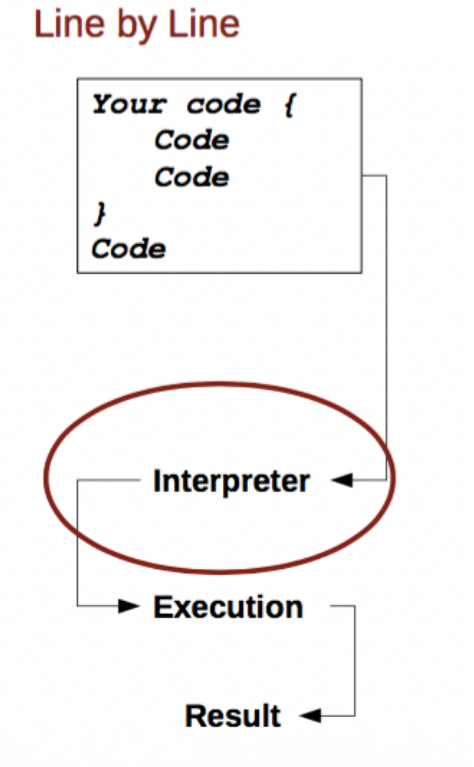
- succinct yet very readable syntax
- great introspection (saves tons of time)
- great standard library
  - built-in data structure (Map, Immutable List)
  - language batteries included
- interactive shell

compiled vs interpreted



pros
- faster execution
- can produce a distributable executable standalone file

cons
- more complicated to build (many files)
- user has to administrate memory usage



pros
- steep learning curve
- takes automatically care of memory usage
- allows fast prototyping

cons
- usually slower
- does not produce standalone programs

Anaconda Python Distribution

- we use Python 3 included in the Anaconda distribution by Continuum Analytics.
- Anaconda is a completely free enterprise-ready Python distribution for large-scale data processing, predictive analytics, and scientific computing.
- apart from that, Anaconda ships with easy-to-use installers for almost every platform, that would drastically reduce the burden of setting up the environment (exp. on Windows)

getting ~~Anaconda~~ Miniconda

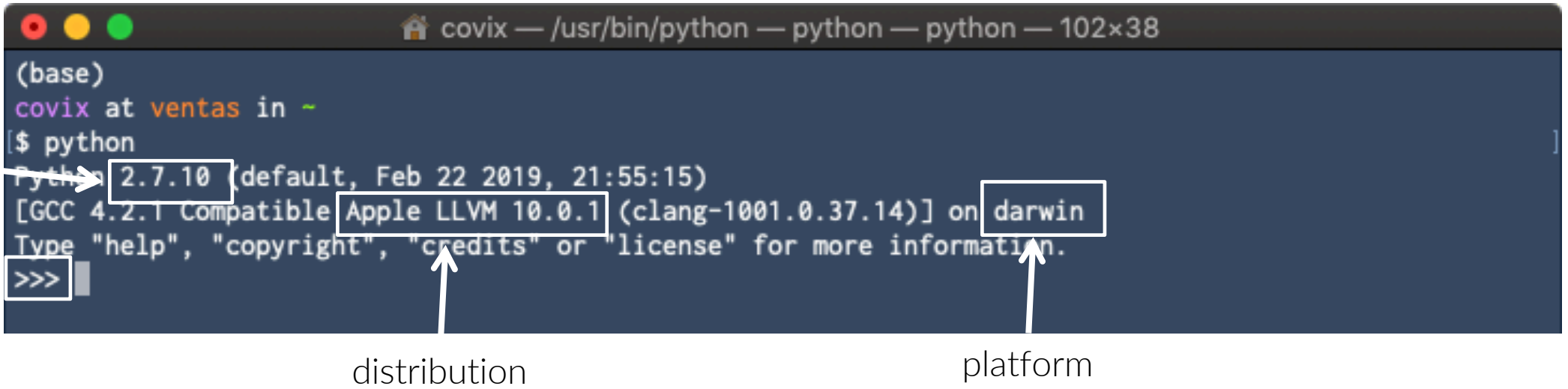
- https://docs.conda.io/en/latest/miniconda.html

- instead of installing the full Anaconda distribution, we will save space installing the Miniconda version
  - conda
  - Python
  - pip
  - and a few others
- `conda install` command installs 720+ additional conda packages

and now?

- type `python` in a shell

read–eval–print loop (REPL)



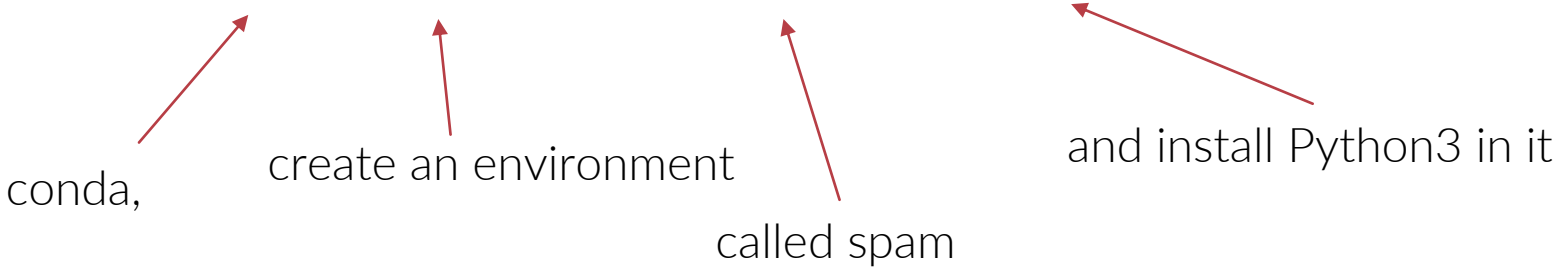
our first Python program

- open a text editor
- type `print("Hello World!")`
- save the file as `hello_world.py`
- now in the shell type `python hello_world.py`

- what happened?
- how long compared to other languages?

(conda) environment(s)

- the Python standard library is great, but it's not enough
- Python libraries can be easily installed with different package managers (`pip`, `conda`, etc.)
- projects are usually organized in different environments (with related packages & versions)
- let's run `conda create --name spam python=3`



- and now, `activate` it!

time to become pythonist

let’s meet variables, types, and the Zen

variables

```
message = "Hello Python world!"
print(message)
```

```
Hello Python world!
```

```
message = "Hello Python world!"
print(message)

message = "Python is my favorite language!"
print(message)
```

```
Hello Python world!
Python is my favorite language!
```

naming rules

- variables can only contain **letters**, **numbers**, and **underscores**.
  - names can start with letters or underscores, but **not** with numbers.
- spaces are not allowed in variable names, use **underscores**!
- you cannot use **Python keywords** as names
- variable names should be descriptive

strings

```
my_string = "This is a double-quoted string."
my_string = 'This is a single-quoted string.'
```

single or double quote!

```
quote = "Linus Torvalds once said, 'Any program is only as good as it is useful.'"
```

it's easy to transform a `string`!

```
first_name = 'eric'

print(first_name)
print(first_name.title())
```

```
first_name = 'ada'
last_name = 'lovelace'

full_name = first_name + ' ' + last_name

print(full_name.title())
```

```
name = ' eric '

print('-' + name.lstrip() + '-')
print('-' + name.rstrip() + '-')
print('-' + name.strip() + '-')
```

strings

```
my_string = "This is a double-quoted string."
my_string = 'This is a single-quoted string.'
```

single or double quote!

```
quote = "Linus Torvalds once said, 'Any program is only as good as it is useful.'"
```

it's easy to transform a `string`!

```
first_name = 'eric'

print(first_name)
print(first_name.title())
```

```
eric
Eric
```

```
first_name = 'ada'
last_name = 'lovelace'

full_name = first_name + ' ' + last_name

print(full_name.title())
```

```
Ada Lovelace
```

```
name = ' eric '

print('-' + name.lstrip() + '-')
print('-' + name.rstrip() + '-')
print('-' + name.strip() + '-')
```

```
-eric -
- eric-
-eric-
```

numbers

integers (int)

```
print(3+2)
```

```
print(3-2)
```

```
print(3*2)
```

```
print(3**2)
```

```
standard_order = 2+3*4
print(standard_order)
```

```
my_order = (2+3)*4
print(my_order)
```

floating points (float)

```
print(0.1+0.1)
```

```
print(0.1+0.2)
```

```
# Python 3.3
print(4/2)
```

```
# Python 3.3
print(3/2)
```

numbers

integers (`int`)

```
print(3+2)
```

```
5
```

```
print(3-2)
```

```
1
```

```
print(3*2)
```

```
6
```

```
print(3**2)
```

```
9
```

```
standard_order = 2+3*4
print(standard_order)
```

```
14
```

```
my_order = (2+3)*4
print(my_order)
```

```
20
```

pretty similar to other languages!

floating points (`float`)

```
print(0.1+0.1)
```

```
0.2
```

```
print(0.1+0.2)
```

```
0.30000000000000004
```

different from other languages?

```
# Python 3.3
print(4/2)
```

```
2.0
```

```
# Python 3.3
print(3/2)
```

```
1.5
```

comments

What makes a good comment?

- It is **short** and to the **point**, but a complete thought
- It **explains** your **thinking** for then you return to the **code later**
- It **explains** your **thinking** to **others** who will **work** with your **code**
- It **explains** particularly difficult sections of **code** in **detail**.

```
# This line is a comment.
print("This line is not a comment, it is code.")
```

```
This line is not a comment, it is code.
```

When should you write comments?

- When you have to **think** about code **before writing** it.
- When you are likely to **forget** later how you **approached** a problem.
- When there is **more** than one **way** to **solve** a problem.
- When **others** are **unlikely** to **anticipate** your way of **thinking**

the Zen of Python

aka `import this`

```
Beautiful is better than ugly.
```

Python programmers recognize that good code can actually be beautiful

```
Explicit is better than implicit.
```

it's always better to be clear

```
Simple is better than complex.
Complex is better than complicated.
```

keep it simple if you can; if not, do not complicate stuff even more

```
Readability counts.
```

Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. Code for readability.
- John F. Woods, 1991

```
There should be one-- and preferably only one --obvious way to do it.
```

```
Now is better than never.
```

your code cannot be perfect, but it doesn't mean that you cannot make it work, for now

exercises! #1

First Name Cases

- Store your first name, in lowercase, in a variable.
- Using that one variable, print your name in lowercase, Titlecase, and UPPERCASE.

Order of Operations

- Find a calculation whose result depends on the order of operations.
- Print the result of this calculation using the standard order of operations.
- Use parentheses to force a nonstandard order of operations. Print the result of this calculation.

Long Decimals

- On paper, 0.1+0.2=0.3. But you have seen that in Python, 0.1+0.2=0.30000000000000004.
- Find at least one other calculation that results in a long decimal like this.

exercises! #1

First Name Cases

- Store your first name, in lowercase, in a variable.
- Using that one variable, print your name in lowercase, Titlecase, and UPPERCASE.

Order of Operations

- Find a calculation whose result depends on the order of operations.
- Print the result of this calculation using the standard order of operations.
- Use parentheses to force a nonstandard order of operations. Print the result of this calculation.

Long Decimals

- On paper, 0.1+0.2=0.3. But you have seen that in Python, 0.1+0.2=0.30000000000000004.
- Find at least one other calculation that results in a long decimal like this.

+ bonus question:
can you multiply a `string` by an `integer`?

a word about types

dynamic typing

- i.e., it is not required to specify the types of variables/functions
- it is automatically inferred by operations

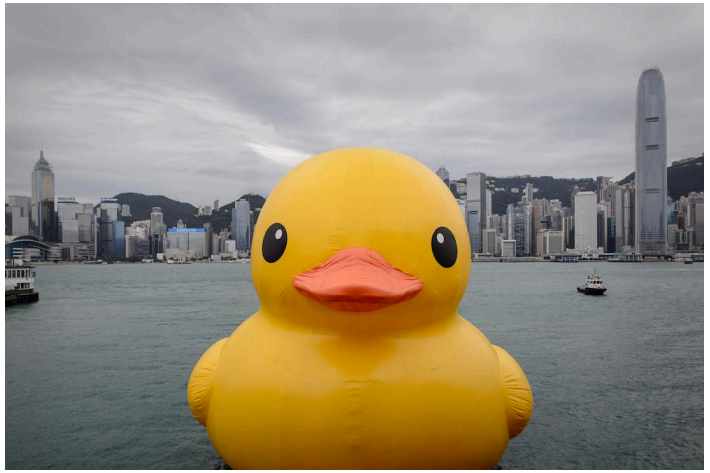
strong typing

- i.e., once the type has been inferred, it cannot change without explicit **cast**

but also...

duck typing!

- if it **walks** like a duck
- and it **quacks** like a duck
- then it must be a **duck**



logical tests

- wait, did we say anything about the `bool` type?
- no, but `bool` corresponds to `int` in Python
  - in addition there are `True` and `False` (as we've already seen)
- we can also use the next conditional operators to compare variables
  - equality (==)
  - inequality (!=)
  - other inequalities
    - greater than (>)
    - greater than or equal to (>=)
    - less than (<)
    - less than or equal to (<=)
- remember: equality means having the same value!

```
5 == 5
```

```
3 == 5
```

```
5 == 5.0
```

```
'eric' == 'eric'
```

```
'Eric' == 'eric'
```

```
'Eric'.lower() == 'eric'.lower()
```

```
'5' == str(5)
```

logical tests

- wait, did we say anything about the `bool` type?
- no, but `bool` corresponds to `int` in Python
  - in addition there are `True` and `False` (as we've already seen)
- we can also use the next conditional operators to compare variables
  - equality (==)
  - inequality (!=)
  - other inequalities
    - greater than (>)
    - greater than or equal to (>=)
    - less than (<)
    - less than or equal to (<=)
- remember: equality means having the same value!

```
5 == 5
```
Out[3]: True

```
3 == 5
```
Out[4]: False

```
5 == 5.0
```
Out[24]: True

```
'eric' == 'eric'
```
Out[8]: True

```
'Eric' == 'eric'
```
Out[9]: False

```
'Eric'.lower() == 'eric'.lower()
```
Out[10]: True

```
'5' == str(5)
```
Out[12]: True

the if, the else and the.. offside

- an `if` statement tests for a condition, and then responds to that condition.
- if the condition is `true`, then whatever action is listed next gets carried out.
- you can test for multiple conditions at the same time, and respond appropriately to each condition.

```
>>> its_raining = True
>>> if its_raining:
...     print("It's raining!")
...
It's raining!
>>> its_raining = False
>>> if its_raining:
...     print("It's raining!")
...
>>>
```

what about this?

the offside rule is used to decide what's inside the block

it applies to **all** blocks! conditionals, loops and functions!

the if, the else and the.. offside

- an `if` statement tests for a condition, and then responds to that condition.
- if the condition is `true`, then whatever action is listed next gets carried out.
- you can test for multiple conditions at the same time, and respond appropriately to each condition.

```
>>> its_raining = True
>>> if its_raining:
...     print("It's raining!")
...
It's raining!
>>> its_raining = False
>>> if its_raining:
...     print("It's raining!")
...
>>>
```

what about this?

the offside rule is used to decide what's inside the block

it applies to **all** blocks! conditionals, loops and functions!

```
if word == "hi":
    print("Hi to you too!")
else:
    if word == "hello":
        print("Hello hello!")
    else:
        if word == "howdy":
            print("Howdyyyy!")
        else:
            if word == "hey":
                print("Hey hey hey!")
            else:
                if word == "gday m8":
                    print("Gday 4 u 2!")
                else:
                    print("I don't know what", word, "means.")
```

```
if word == "hi":
    print("Hi to you too!")
elif word == "hello":
    print("Hello hello!")
elif word == "howdy":
    print("Howdyyyy!")
elif word == "hey":
    print("Hey hey hey!")
elif word == "gday m8":
    print("Gday 4 u 2!")
else:
    print("I don't know what", word, "means.")
```

never forget about `elif`!

but, what is True?

1
```
if 0:
    print("This evaluates to True.")
else:
    print("This evaluates to False.")
```

2
```
if 1:
    print("This evaluates to True.")
else:
    print("This evaluates to False.")
```

3
```
if 1253756:
    print("This evaluates to True.")
else:
    print("This evaluates to False.")
```

3
```
if -1:
    print("This evaluates to True.")
else:
    print("This evaluates to False.")
```

4
```
if '':
    print("This evaluates to True.")
else:
    print("This evaluates to False.")
```

5
```
if 'hello':
    print("This evaluates to True.")
else:
    print("This evaluates to False.")
```

6
```
if ' ':
    print("This evaluates to True.")
else:
    print("This evaluates to False.")
```

7
```
if None:
    print("This evaluates to True.")
else:
    print("This evaluates to False.")
```

but, what is True?

1

```
if 0:
    print("This evaluates to True.")
else:
    print("This evaluates to False.")
```

This evaluates to False.

2

```
if 1:
    print("This evaluates to True.")
else:
    print("This evaluates to False.")
```

This evaluates to True.

3

```
if 1253756:
    print("This evaluates to True.")
else:
    print("This evaluates to False.")
```

This evaluates to True.

3

```
if -1:
    print("This evaluates to True.")
else:
    print("This evaluates to False.")
```

This evaluates to True.

4

```
if '':
    print("This evaluates to True.")
else:
    print("This evaluates to False.")
```

This evaluates to False.

5

```
if 'hello':
    print("This evaluates to True.")
else:
    print("This evaluates to False.")
```

This evaluates to True.

6

```
if ' ':
    print("This evaluates to True.")
else:
    print("This evaluates to False.")
```

This evaluates to True.

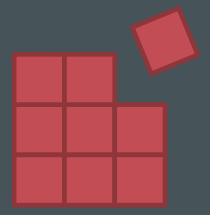
7

```
if None:
    print("This evaluates to True.")
else:
    print("This evaluates to False.")
```

This evaluates to False.

built-in data structures

iterables: lists, tuples, sets, dictionaries



[lists]

- a list is a **collection** of **items**, that is stored in a variable.
- the **items** should be **related** in some way
  - but there are **no restrictions** on what can be stored in a list.
- since lists are **collection** of objects, it is **good practice** to give them a **plural name**.
  - if each item in your list is a car, call the list 'cars'.

```
dogs = ['border collie', 'australian cattle dog', 'labrador retriever']

dog = dogs[0]
print(dog.title())
```

```
Border Collie
```

```
dog = dogs[-1]
print(dog.title())
```

```
Labrador Retriever
```

```
dog = dogs[-2]
print(dog.title())
```

```
Australian Cattle Dog
```

[lists]

- a list is a **collection** of **items**, that is stored in a variable.
- the **items** should be **related** in some way
  - but there are **no restrictions** on what can be stored in a list.
- since lists are **collection** of objects, it is **good practice** to give them a **plural name**.
  - if each item in your list is a car, call the list 'cars'.

```
dogs = ['border collie', 'australian cattle dog', 'labrador retriever']

dog = dogs[0]
print(dog.title())
```

```
Border Collie
```

```
dog = dogs[-1]
print(dog.title())
```

```
Labrador Retriever
```

```
dog = dogs[-2]
print(dog.title())
```

```
Australian Cattle Dog
```

square brackets define lists

```
students = ['bernice', 'aaron', 'cody']

for student in students:
    print("Hello, " + student.title() + "!")
```

```
Hello, Bernice!
Hello, Aaron!
Hello, Cody!
```

`for` is a keyword for the for (-each) loop

`student` is a temporary variable

the nested block is create with the offside rule

enumerate a list

- what if you want to print the index and the value at that index in a list?
- just use `value[i]`, right?
- or maybe use the function `enumerate`!

```
dogs = ['border collie', 'australian cattle dog', 'labrador retriever']

print("Results for the dog show are as follows:\n")
for index, dog in enumerate(dogs):
    place = str(index)
    print("Place: " + place + " Dog: " + dog.title())
```

```
Results for the dog show are as follows:

Place: 0 Dog: Border Collie
Place: 1 Dog: Australian Cattle Dog
Place: 2 Dog: Labrador Retriever
```

common list operations

Modifying elements in a list

You can change the value of any element in a list if you know the position of that item.

```
dogs = ['border collie', 'australian cattle dog', 'labrador retriever']

dogs[0] = 'australian shepherd'
print(dogs)
```

```
['australian shepherd', 'australian cattle dog', 'labrador retriever']
```

Testing whether an item is in a list

```
print('australian cattle dog' in dogs)
print('poodle' in dogs)
```

```
True
False
```

Finding the length of a list

```
usernames = ['bernice', 'cody', 'aaron']
user_count = len(usernames)

print(user_count)
```

```
3
```

Finding an element in a list

If you want to find out the position of an element in a list, you can use the index() function.

```
dogs = ['border collie', 'australian cattle dog', 'labrador retriever']

print(dogs.index('australian cattle dog'))
```

```
1
```

Looking for a non existing item

```
dogs = ['border collie', 'australian cattle dog', 'labrador retriever']

print(dogs.index('poodle'))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-13-a9e05e37e8df> in <module>()
      1 dogs = ['border collie', 'australian cattle dog', 'labrador retriever']
      2
----> 3 print(dogs.index('poodle'))

ValueError: 'poodle' is not in list
```

more common list operations

Appending items to the end of a list

```
dogs = ['border collie', 'australian cattle dog', 'labrador retriever']
dogs.append('poodle')

for dog in dogs:
    print(dog.title() + "s are cool.")
```

```
Border Collies are cool.
Australian Cattle Dogs are cool.
Labrador Retrievers are cool.
Poodles are cool.
```

Inserting items into a list

```
dogs = ['border collie', 'australian cattle dog', 'labrador retriever']
dogs.insert(1, 'poodle')

print(dogs)
```

```
['border collie', 'poodle', 'australian cattle dog', 'labrador retriever']
```

Sorting a List

```
students = ['bernice', 'aaron', 'cody']

# Put students in alphabetical order.
students.sort()

# Display the list in its current order.
print("Our students are currently in alphabetical order.")
for student in students:
    print(student.title())
```

```
Our students are currently in alphabetical order.
Aaron
Bernice
Cody
```

sorted() vs. *sort()*

```
# Display students in alphabetical order, but keep the original order.
print("Here is the list in alphabetical order:")
for student in sorted(students):
    print(student.title())
```

```
Here is the list in alphabetical order:
Aaron
Bernice
Cody
```

`sorted` doesn't sort the list in-place!

more more common list operation

Removing items by position

```
dogs = ['border collie', 'australian cattle dog', 'labrador retriever']
# Remove the first dog from the list.
del dogs[0]

print(dogs)
```

```
['australian cattle dog', 'labrador retriever']
```

```
letters = ['a', 'b', 'c', 'a', 'b', 'c']
# Remove the letter a from the list.
letters.remove('a')

print(letters)
```

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

Popping items from a list

```
dogs = ['border collie', 'australian cattle dog', 'labrador retriever']
last_dog = dogs.pop()

print(last_dog)
print(dogs)
```

```
['border collie', 'labrador retriever']
```

```
dogs = ['border collie', 'australian cattle dog', 'labrador retriever']
first_dog = dogs.pop(0)

print(first_dog)
print(dogs)
```

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

exercises! #2

Working List

- Make a list that includes four careers, such as 'programmer' and 'truck driver'.
- Use the *list.index()* function to find the index of one career in your list.
- Use the *in* function to show that this career is in your list.
- Use the *append()* function to add a new career to your list.
- Use the *insert()* function to add a new career at the beginning of the list.
- Use a loop to show all the careers in your list.

remember: save everything in scripts files (`.py`) and execute them with

```
python working_list.py
```

Ordered Working List

- Start with the list you created in *Working List*.
- You are going to print out the list in a number of different orders.
- Each time you print the list, use a for loop rather than printing the raw list.
- Print a message each time telling us what order we should see the list in.
  - Print the list in its original order.
  - Print the list in alphabetical order.
  - Print the list in its original order.
  - Print the list in reverse alphabetical order.
  - Print the list in its original order.
  - Print the list in the reverse order from what it started.
  - Print the list in its original order
  - Permanently sort the list in alphabetical order, and then print it out.
  - Permanently sort the list in reverse alphabetical order, and then print it out.

note: some of you may want to use Python `functions`: you can if you're brave enough!

slicing a list[:]

- a list is a collection of items and we should be able to get any subset
- we can use the notation `list[start_index:end_index]` to select all the elements from index `start_index` (included) to `end_index` (excluded)

```
usernames = ['bernice', 'cody', 'aaron', 'ever', 'dalia']

# Grab the first three users in the list.
first_batch = usernames[0:3]

for user in first_batch:
    print(user.title())
```

```
Bernice
Cody
Aaron
```

```
# Grab the first three users in the list.
first_batch = usernames[:3]

for user in first_batch:
    print(user.title())
```

```
Bernice
Cody
Aaron
```

```
# Grab all users from the third to the end.
end_batch = usernames[2:]

for user in end_batch:
    print(user.title())
```

```
Aaron
Ever
Dalia
```

deep vs shallow copy

- `usernames` is not really a `list`: under the hood is a pointer to a list in memory
- if we want to make a deep copy of the list, we can slice all the elements out of it

```
usernames = ['bernice', 'cody', 'aaron', 'ever', 'dalia']

# Make a copy of the list.
copied_usernames = usernames[:]
print("The full copied list:\n\t", copied_usernames)

# Remove the first two users from the copied list.
del copied_usernames[0]
del copied_usernames[0]
print("\nTwo users removed from copied list:\n\t", copied_usernames)

# The original list is unaffected.
print("\nThe original list:\n\t", usernames)
```

```
The full copied list:
        ['bernice', 'cody', 'aaron', 'ever', 'dalia']

Two users removed from copied list:
        ['aaron', 'ever', 'dalia']

The original list:
        ['bernice', 'cody', 'aaron', 'ever', 'dalia']
```

numerical lists

aka do not expect anything special, apart from `range`, `min`, `max` and `sum`

```
# Print the first ten numbers.
for number in range(1,11):
    print(number)
```

```
1
2
3
4
5
6
7
8
9
10
```

mini exercise:
can you type just `range(11)`?

```
# Print the first ten odd numbers.
for number in range(1,21,2):
    print(number)
```

```
1
3
5
7
9
11
13
15
17
19
```

mini exercise:
can you type just `range(21,2)`?

```
# Create a list of the first ten numbers.
numbers = list(range(1,11))
print(numbers)
```

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

mini exercise:
what if you don't use `list`?

```
ages = [23, 16, 14, 28, 19, 11, 38]

youngest = min(ages)
oldest = max(ages)
total_years = sum(ages)

print("Our youngest reader is " + str(youngest) + " years old.")
print("Our oldest reader is " + str(oldest) + " years old.")
print("Together, we have " + str(total_years) + " years worth of life experience.")
```

```
Our youngest reader is 11 years old.
Our oldest reader is 38 years old.
Together, we have 149 years worth of life experience.
```

[list comprehensions]

```
# Store the first ten square numbers in a list.
# Make an empty list that will hold our square numbers.
squares = []

# Go through the first ten numbers, square them, and add them to our list.
for number in range(1,11):
    squares.append(number**2)

# Show that our list is correct.
for square in squares:
    print(square)
```

```
1
4
9
16
25
36
49
64
81
100
```

it allow us to collapse the first three lines of code into one line:

```
# Store the first ten square numbers in a list.
squares = [number**2 for number in range(1,11)]

# Show that our list is correct.
for square in squares:
    print(square)
```

```
1
4
9
16
25
36
49
64
81
100
```

```
# Consider some students.
students = ['bernice', 'aaron', 'cody']

# Let's turn them into great students.
great_students = [student.title() + " the great!" for student in students]

# Let's greet each great student.
for great_student in great_students:
    print("Hello, " + great_student)
```

```
Hello, Bernice the great!
Hello, Aaron the great!
Hello, Cody the great!
```

exercises! #3

Multiples of Ten

- Make a list of the first ten multiples of ten (10, 20, 30... 90, 100).

Working Backwards

- Write out the following code without using a list comprehension:

  plus_thirteen = [number + 13 for number in range(1,11)]

“strings” (again?)

we can use them as `list`s, and we can convert them to `list`s

```
message = "Hello!"

for letter in message:
    print(letter)
```

```
H
e
l
l
o
!
```

```
message = "Hello world!"

message_list = list(message)
print(message_list)
```

```
['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '!']
```

```
message = "Hello World!"
first_char = message[0]
last_char = message[-1]

print(first_char, last_char)
```

```
('H', '!')
```

```
message = "Hello World!"
first_three = message[:3]
last_three = message[-3:]

print(first_three, last_three)
```

```
('Hel', 'ld!')
```

```
message = "I like cats and dogs."
dog_present = 'dog' in message
print(dog_present)
```

```
True
```

```
message = "I like cats and dogs, but I'd much rather own a dog."
last_dog_index = message.rfind('dog')
print(last_dog_index)
```

```
48
```

```
message = "I like cats and dogs, but I'd much rather own a dog."
dog_index = message.find('dog')
print(dog_index)
```

```
16
```


“strings” (again again?)

Replacing substrings

```
message = "I like cats and dogs, but I'd much rather own a dog."
message = message.replace('dog', 'snake')
print(message)
```

```
I like cats and snakes, but I'd much rather own a snake.
```

Counting substrings

```
message = "I like cats and dogs, but I'd much rather own a dog."
number_dogs = message.count('dog')
print(number_dogs)
```

```
2
```

Splitting strings

```
message = "I like cats and dogs, but I'd much rather own a dog."
words = message.split(' ')
print(words)
```

```
['I', 'like', 'cats', 'and', 'dogs,', 'but', "I'd", 'much', 'rather', 'own', 'a', 'dog.']
```

```
animals = "dog, cat, tiger, mouse, liger, bear"

# Rewrite the string as a list, and store it in the same variable
animals = animals.split(',')
print(animals)
```

```
['dog', ' cat', ' tiger', ' mouse', ' liger', ' bear']
```

exercises! #4

Finding Python

- Store a sentence in a variable, making sure you use the word *Python* at least twice in the sentence.
- Use the *in* keyword to prove that the word *Python* is actually in the sentence.
- Use the *find()* function to show where the word *Python* first appears in the sentence.
- Use the *rfind()* function to show the last place *Python* appears in the sentence.
- Use the *count()* function to show how many times the word *Python* appears in your sentence.
- Use the *split()* function to break your sentence into a list of words. Print the raw list, and use a loop to print each word on its own line.
- Use the *replace()* function to change *Python* to *Ruby* in your sentence.

Challenges

Counting DNA Nucleotides

- Project Rosalind is a problem set based on biotechnology concepts. It is meant to show how programming skills can help solve problems in genetics and biology.
- If you have understood this section on strings, you have enough information to solve the first problem in Project Rosalind, Counting DNA Nucleotides. Give the sample problem a try.
- If you get the sample problem correct, log in and try the full version of the problem!

Transcribing DNA into RNA

- You also have enough information to try the second problem, Transcribing DNA into RNA. Solve the sample problem.
- If you solved the sample problem, log in and try the full version!

Complementing a Strand of DNA

- You guessed it, you can now try the third problem as well: Complementing a Strand of DNA. Try the sample problem, and then try the full version if you are successful.

`(tuples,)`

aka *immutable* `list`

- **lists** are quite **dynamic**; they can grow and shrink
- you can modify any element you want to in a list.
- **sometimes** we **like** this behaviour, but **other times** we may want to ensure that **no user** or **no** part of a **program** can **change** a **list**.
- that's what **tuples** are for

```
colors = ('red', 'green', 'blue')
print("The first color is: " + colors[0])

print("\nThe available colors are:")
for color in colors:
    print("- " + color)
```

```
The first color is: red

The available colors are:
- red
- green
- blue
```

```
colors = ('red', 'green', 'blue')
colors.append('purple')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-37-ed1dbff53ab2> in <module>()
      1 colors = ('red', 'green', 'blue')
----> 2 colors.append('purple')

AttributeError: 'tuple' object has no attribute 'append'
```

while loop

General syntax

```
# Set an initial condition.
game_active = True

# Set up the while loop.
while game_active:
    # Run the game.
    # At some point, the game ends and game_active will be set to False.
    #   When that happens, the loop will stop executing.

# Do anything else you want done after the loop runs.
```

- Every while loop needs an initial condition that starts out true.
- The `while` statement includes a condition to test.
- All of the code in the loop will run as long as the condition remains true.
- As soon as something in the loop changes the condition such that the test no longer passes, the loop stops executing.
- Any code that is defined after the loop will run at this point.

more while loop

```
# The player's power starts out at 5.
power = 5

# The player is allowed to keep playing as long as their power is over 0.
while power > 0:
    print("You are still playing, because your power is %d." % power)
    # Your game code would go here, which includes challenges that make it
    #   possible to lose power.
    # We can represent that by just taking away from the power.
    power = power - 1

print("\nOh no, your power dropped to 0! Game Over.")
```

```
You are still playing, because your power is 5.
You are still playing, because your power is 4.
You are still playing, because your power is 3.
You are still playing, because your power is 2.
You are still playing, because your power is 1.

Oh no, your power dropped to 0! Game Over.
```

historical & mathematical exercise!

Gaussian Addition

This challenge is inspired by a story about the mathematician Carl Frederich Gauss. As the story goes, when young Gauss was in grade school his teacher got mad at his class one day.

"I'll keep the lot of you busy for a while", the teacher said sternly to the group. "You are to add the numbers from 1 to 100, and you are not to say a word until you are done."

The teacher expected a good period of quiet time, but a moment later our mathematician-to-be raised his hand with the answer. "It's 5050!" Gauss had realized that if you list all the numbers from 1 to 100, you can always match the first and last numbers in the list and get a common answer:

```
1, 2, 3, ..., 98, 99, 100
1 + 100 = 101
2 + 99 = 101
3 + 98 = 101
```

Gauss realized there were exactly 50 pairs of numbers in the range 1 to 100, so he did a quick calculation: 50 * 101 = 5050.

- Write a program that passes a list of numbers to a function.
  - The function should use a while loop to keep popping the first and last numbers from the list and calculate the sum of those two numbers.
  - The function should print out the current numbers that are being added, and print their partial sum.
  - The function should keep track of how many partial sums there are.
  - The function should then print out how many partial sums there were.
  - The function should perform Gauss' multiplication, and report the final answer.
- Prove that your function works, by passing in the range 1-100, and verifying that you get 5050.
  - `gauss_addition(list(range(1,101)))`
- Your function should work for any set of consecutive numbers, as long as that set has an even length.
  - Bonus: Modify your function so that it works for any set of consecutive numbers, whether that set has an even or odd length.

{dictionaries: ‘what are they’}

- dictionaries are a way to store information that is connected in some way.
- dictionaries store information in key-value pairs,
  - any piece of information is connected to at least one other piece of information.
- dictionaries do not store their information in any particular order

```
dictionary_name = {key_1: value_1,
                   key_2: value_2,
                   key_3: value_3,
                   }
```

{dictionaries: ‘how to iterate’}

```
python_words = {'list': 'A collection of values that are not connected, but have an order.',
                'dictionary': 'A collection of key-value pairs.',
                'function': 'A named set of instructions that defines a set of actions in Python.',
                }

# Print out the items in the dictionary.
for word, meaning in python_words.items():
    print("\nWord: %s" % word)
    print("Meaning: %s" % meaning)
```

```
Word: list
Meaning: A collection of values that are not connected, but have an order.

Word: dictionary
Meaning: A collection of key-value pairs.

Word: function
Meaning: A named set of instructions that defines a set of actions in Python.
```

{more: dictionaries}

```
# Create an empty dictionary.
python_words = {}

# Fill the dictionary, pair by pair.
python_words['list'] ='A collection of values that are not connected, but have an order.'
python_words['dictionary'] = 'A collection of key-value pairs.'
python_words['function'] = 'A named set of instructions that defines a set of actions in Python.'

# Print out the items in the dictionary.
for word, meaning in python_words.items():
    print("\nWord: %s" % word)
    print("Meaning: %s" % meaning)
```

```
Word: function
Meaning: A named set of instructions that defines a set of actions in Python.

Word: list
Meaning: A collection of values that are not connected, but have an order.

Word: dictionary
Meaning: A collection of key-value pairs.
```

```
# Remove the word 'list' and its meaning.
del python_words['list']
```

```
# We have a spelling mistake!
python_words = {'lisst': 'A collection of values that are not connected, but have an order.'}

# Create a new, correct key, and connect it to the old value.
#  Then delete the old key.
python_words['list'] = python_words['lisst']
del python_words['lisst']

# Print the dictionary, to show that the key has changed.
print(python_words)
```

```
{'list': 'A collection of values that are not connected, but have an order.'}
```

looping through dictionaries

accessing to key-value pairs

```
my_dict = {'key_1': 'value_1',
    'key_2': 'value_2',
    'key_3': 'value_3',
    }

for key, value in my_dict.items():
    print('\nKey: %s' % key)
    print('Value: %s' % value)
```

```
Key: key_1
Value: value_1

Key: key_3
Value: value_3

Key: key_2
Value: value_2
```

accessing to values

```
my_dict = {'key_1': 'value_1',
    'key_2': 'value_2',
    'key_3': 'value_3',
    }

for value in my_dict.values():
    print('Value: %s' % value)
```

```
Value: value_1
Value: value_3
Value: value_2
```

accessing to keys

```
my_dict = {'key_1': 'value_1',
    'key_2': 'value_2',
    'key_3': 'value_3',
    }

for key in my_dict.keys():
    print('Key: %s' % key)
```

```
Key: key_1
Key: key_3
Key: key_2
```

exercises! #5

Mountain Heights

- Wikipedia has a list of the tallest mountains in the world, with each mountain's elevation. Pick five mountains from this list.
  - Create a dictionary with the mountain names as keys, and the elevations as values.
  - Print out just the mountains' names, by looping through the keys of your dictionary.
  - Print out just the mountains' elevations, by looping through the values of your dictionary.
  - Print out a series of statements telling how tall each mountain is: "Everest is 8848 meters tall."
- Revise your output, if necessary.
  - Make sure there is an introductory sentence describing the output for each loop you write.
  - Make sure there is a blank line between each group of statements.

Mountain Heights 2

- Revise your final output from Mountain Heights, so that the information is listed in alphabetical order by each mountain's name.
  - That is, print out a series of statements telling how tall each mountain is: "Everest is 8848 meters tall."
  - Make sure your output is in alphabetical order.

functions()

- functions are a set of actions that we group together, and give a name to.
- you have already used a number of functions from the core Python language, such as **`string.title()`** and **`list.sort()`**.
- we can define our own functions, which allows us to "teach" Python new behaviour.

```
# Let's define a function.
def function_name(argument_1, argument_2):
        # Do whatever we want this function to do,
        #  using argument_1 and argument_2

# Use function_name to call the function.
function_name(value_1, value_2)
```

functions(more)

- functions are a set of actions that we group together, and give a name to.
- you have already used a number of functions from the core Python language, such as **`string.title()`** and **`list.sort()`**.
- we can define our own functions, which allows us to "teach" Python new behaviour.
- by default functions return **`None`**, we can change this behaviour using the **`return`** keyword

```
# Let's define a function.
def function_name(argument_1, argument_2):
        # Do whatever we want this function to do,
        #  using argument_1 and argument_2

# Use function_name to call the function.
function_name(value_1, value_2)
```

```
def get_number_word(number):
    # Takes in a numerical value, and returns
    #  the word corresponding to that number.
    if number == 0:
        return 'zero'
    elif number == 1:
        return 'one'
    elif number == 2:
        return 'two'
    elif number == 3:
        return 'three'
    else:
        return "I'm sorry, I don't know that number."
```

function(arguments)

default arguments

```
def thank_you(name='everyone'):
    # This function prints a two-line personalized thank you message.
    #  If no name is passed in, it prints a general thank you message
    #  to everyone.
    print("\nYou are doing good work, %s!" % name)
    print("Thank you very much for your efforts on this project.")

thank_you('Adriana')
thank_you('Billy')
thank_you('Caroline')
thank_you()
```

```
You are doing good work, Adriana!
Thank you very much for your efforts on this project.

You are doing good work, Billy!
Thank you very much for your efforts on this project.

You are doing good work, Caroline!
Thank you very much for your efforts on this project.

You are doing good work, everyone!
Thank you very much for your efforts on this project.
```

keyword arguments

```
def describe_person(first_name, last_name, age):
    # This function takes in a person's first and last name,
    #  and their age.
    # It then prints this information out in a simple format.
    print("First name: %s" % first_name.title())
    print("Last name: %s" % last_name.title())
    print("Age: %d\n" % age)

describe_person(age=71, first_name='brian', last_name='kernighan')
describe_person(age=70, first_name='ken', last_name='thompson')
describe_person(age=68, first_name='adele', last_name='goldberg')
```

```
First name: Brian
Last name: Kernighan
Age: 71

First name: Ken
Last name: Thompson
Age: 70

First name: Adele
Last name: Goldberg
Age: 68
```

*arguments of variable length

*args lets you accept a variable number of arguments

```
def example_function(arg_1, arg_2, *arg_3):
    # Let's look at the argument values.
    print('\narg_1:', arg_1)
    print('arg_2:', arg_2)
    print('arg_3:', arg_3)

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

```
arg_1: 1
arg_2: 2
arg_3: ()

arg_1: 1
arg_2: 2
arg_3: (3,)

arg_1: 1
arg_2: 2
arg_3: (3, 4)

arg_1: 1
arg_2: 2
arg_3: (3, 4, 5)
```

you can consider *args as a list

```
def example_function(arg_1, arg_2, *arg_3):
    # Let's look at the argument values.
    print('\narg_1:', arg_1)
    print('arg_2:', arg_2)
    for value in arg_3:
        print('arg_3 value:', value)

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

```
arg_1: 1
arg_2: 2

arg_1: 1
arg_2: 2
arg_3 value: 3

arg_1: 1
arg_2: 2
arg_3 value: 3
arg_3 value: 4

arg_1: 1
arg_2: 2
arg_3 value: 3
arg_3 value: 4
arg_3 value: 5
```

**keyword arguments of variable length

**kwargs lets you accept a variable number of keyword arguments

```
def example_function(arg_1, arg_2, **kwargs):
    # Let's look at the argument values.
    print('\narg_1:', arg_1)
    print('arg_2:', arg_2)
    print('arg_3:', kwargs)

example_function('a', 'b')
example_function('a', 'b', value_3='c')
example_function('a', 'b', value_3='c', value_4='d')
example_function('a', 'b', value_3='c', value_4='d', value_5='e')
```

```
arg_1: a
arg_2: b
arg_3: {}

arg_1: a
arg_2: b
arg_3: {'value_3': 'c'}

arg_1: a
arg_2: b
arg_3: {'value_4': 'd', 'value_3': 'c'}

arg_1: a
arg_2: b
arg_3: {'value_5': 'e', 'value_4': 'd', 'value_3': 'c'}
```

you can consider **kwargs as a dictionary

```
def example_function(arg_1, arg_2, **kwargs):
    # Let's look at the argument values.
    print('\narg_1:', arg_1)
    print('arg_2:', arg_2)
    for key, value in kwargs.items():
        print('arg_3 value:', value)

example_function('a', 'b')
example_function('a', 'b', value_3='c')
example_function('a', 'b', value_3='c', value_4='d')
example_function('a', 'b', value_3='c', value_4='d', value_5='e')
```

```
arg_1: a
arg_2: b

arg_1: a
arg_2: b
arg_3 value: c

arg_1: a
arg_2: b
arg_3 value: d
arg_3 value: c

arg_1: a
arg_2: b
arg_3 value: e
arg_3 value: d
arg_3 value: c
```

class:

- classes are a way of combining information and behaviour.
- let's consider what you'd need to do if you were creating a rocket ship in a game:
  - you'd want to track are the x and y coordinates of the rocket

the objective is not to learn about oop design's principle, but to get used to Python syntax for classes!

```
class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self):
        # Each rocket has an (x,y) position.
        self.x = 0
        self.y = 0

    def move_up(self):
        # Increment the y-position of the rocket.
        self.y += 1

# Create a Rocket object, and have it start to move up.
my_rocket = Rocket()
print("Rocket altitude:", my_rocket.y)

my_rocket.move_up()
print("Rocket altitude:", my_rocket.y)

my_rocket.move_up()
print("Rocket altitude:", my_rocket.y)
```

class constructor

instance method

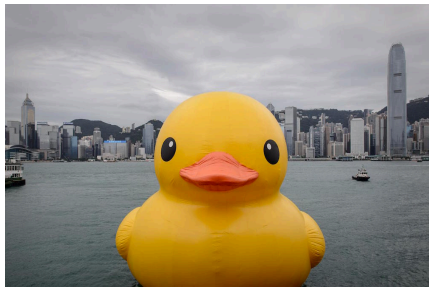
```
Rocket altitude: 0
Rocket altitude: 1
Rocket altitude: 2
```

`self` is the keyword for the instance itself

class(inheritance):

- the class `Rocket` is our base class
- we can create other classes that are based on it
- for example, we can create a reusable rocket, or a `Shuttle`
  - this child class will have access to the `move_rocket` and `get_distance` functions

remember: a `Shuttle` can *quack* like a `Rocket`



```
from math import sqrt

class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self, x=0, y=0):
        # Each rocket has an (x,y) position.
        self.x = x
        self.y = y

    def move_rocket(self, x_increment=0, y_increment=1):
        # Move the rocket according to the paremeters given.
        #  Default behavior is to move the rocket up one unit.
        self.x += x_increment
        self.y += y_increment

    def get_distance(self, other_rocket):
        # Calculates the distance from this rocket to another rocket,
        #  and returns that value.
        distance = sqrt((self.x-other_rocket.x)**2+(self.y-other_rocket.y)**2)
        return distance

class Shuttle(Rocket):
    # Shuttle simulates a space shuttle, which is really
    #  just a reusable rocket.

    def __init__(self, x=0, y=0, flights_completed=0):
        super().__init__(x, y)
        self.flights_completed = flights_completed

shuttle = Shuttle(10,0,3)
print(shuttle)
```

from module `math` import function `sqrt`

```
<__main__.Shuttle object at 0x7f1e62ba6cd0>
```

from module import classes

- we can save the `Rocket` and the `Shuttle` classes in a `module` (python script) called `rocket.py`
- now we can import those classes from the `rocket module` and reuse them

```
from rocket import Rocket, Shuttle

rocket = Rocket()
print("The rocket is at (%d, %d)." % (rocket.x, rocket.y))

shuttle = Shuttle()
print("\nThe shuttle is at (%d, %d)." % (shuttle.x, shuttle.y))
print("The shuttle has completed %d flights." % shuttle.flights_completed)
```

```
The rocket is at (0, 0).

The shuttle is at (0, 0).
The shuttle has completed 0 flights.
```

- wait, `modules`?
- `modules` are nothing more than python scripts containing reusable code (`classes`, `functions`, etc.)
- `modules` are organized in `packages`

open(files)

Writing to a file

Let's create a file and write a hello world to it.

```
>>> with open('hello.txt', 'w') as f:
...     print("Hello World!", file=f)
...
>>>
```

```
>>> lines = []
>>> with open('hello.txt', 'r') as f:
...     for line in f:
...         lines.append(line)
...
>>> lines
['Hello World!\n']
>>>
```

Mode	Short for	Meaning
r	read	Read from an existing file.
w	write	Write to a file. If the file exists, its old content is removed.
a	append	Write to the end of a file, and keep the old content.

time to code!

1. port scanner
2. ip geo location
3. web scraping ws19
4. simple http server
5. wireshark log parser

`port scanner`

- ask the user for a hostname
- extract the ip from the hostname
- ask the user for a port to scan
- create a socket and try to connect

- hint: try the `socket` library

port scanner

- ask the user for a hostname
- extract the ip from the hostname
- ask the user for a port to scan
- create a socket and try to connect

```
t_host = str(input("Enter the host to be scanned: "))
```

- hint: try the `socket` library

port scanner

- ask the user for a hostname
- extract the ip from the hostname
- ask the user for a port to scan
- create a socket and try to connect

```
t_host = str(input("Enter the host to be scanned: "))
```

```
t_ip = socket.gethostbyname(t_host)
```

- hint: try the `socket` library

port scanner

- ask the user for a hostname
- extract the ip from the hostname
- ask the user for a port to scan
- create a socket and try to connect

```
t_host = str(input("Enter the host to be scanned: "))
```

```
t_ip = socket.gethostbyname(t_host)
```

```
t_port = int(input("Enter the port: "))
```

- hint: try the `socket` library

port scanner

- ask the user for a hostname

```
t_host = str(input("Enter the host to be scanned: "))
```

- extract the ip from the hostname

```
t_ip = socket.gethostbyname(t_host)
```

- ask the user for a port to scan

```
t_port = int(input("Enter the port: "))
```

- create a socket and try to connect

```
sock = socket.socket()
res = sock.connect((t_ip, t_port))
print(f"Port {t_port}: Open")
sock.close()
```

- hint: try the `socket` library

ip geo location

- we need the `geoip2==2.9.0` package (`pip install geoip2==2.9.0`) and the GeoLite2 database
- load the database
- query the database
- extract the interesting fields

ip geo location

- we need the `geoip2==2.9.0` package (`pip install geoip2==2.9.0`) and the GeoLite2 database
- load the database

```
reader = geoip2.database.Reader("./GeoLite2-City_20191001/GeoLite2-City.mmdb")
```

- query the database
- extract the interesting fields

ip geo location

- we need the `geoip2==2.9.0` package (`pip install geoip2==2.9.0`) and the GeoLite2 database

- load the database

```
reader = geoip2.database.Reader("./GeoLite2-City_20191001/GeoLite2-City.mmdb")
```

- query the database

```
response = reader.city(ip)
```

- extract the interesting fields

ip geo location

- we need the `geoip2==2.9.0` package (`pip install geoip2==2.9.0`) and the GeoLite2 database

- load the database

```
reader = geoip2.database.Reader("./GeoLite2-City_20191001/GeoLite2-City.mmdb")
```

- query the database

```
response = reader.city(ip)
```

- extract the interesting fields

```
city = response.city.name
country = response.country.name
longitude = response.location.longitude
lat = response.location.latitude
```

web scraping ws19

- we need the `beautifulsoup4` and `requests` packages
- requests the `ws19` programme page
- parse the `html` into a `bs4` object
- extract all the `tr` elements with `uk-alert` and `uk-alert-table` classes
- for each found element print the `h4` and `p` text

web scraping ws19

- we need the `beautifulsoup4` and `requests` packages
- requests the `ws19` programme page

```
r = requests.get(url)
```

- parse the `html` into a `bs4` object
- extract all the `tr` elements with `uk-alert` and `uk-alert-table` classes
- for each found element print the `h4` and `p` text

web scraping ws19

- we need the `beautifulsoup4` and `requests` packages
- requests the `ws19` programme page
- parse the `html` into a `bs4` object
- extract all the `tr` elements with `uk-alert` and `uk-alert-table` classes
- for each found element print the `h4` and `p` text

```
r = requests.get(url)
```

```
soup = BeautifulSoup(r.content)
```

web scraping ws19

- we need the `beautifulsoup4` and `requests` packages

- requests the `ws19` programme page

```
r = requests.get(url)
```

- parse the `html` into a `bs4` object

```
soup = BeautifulSoup(r.content)
```

- extract all the `tr` elements with `uk-alert` and `uk-alert-table` classes

```
events = soup.find_all("tr",
                       {"class": "uk-alert uk-alert-table"})
```

- for each found element print the `h4` and `p` text

web scraping ws19

- we need the `beautifulsoup4` and `requests` packages

- requests the `ws19` programme page

```
r = requests.get(url)
```

- parse the `html` into a `bs4` object

```
soup = BeautifulSoup(r.content)
```

- extract all the `tr` elements with `uk-alert` and `uk-alert-table` classes

```
events = soup.find_all("tr",
                       {"class": "uk-alert uk-alert-table"})
```

- for each found element print the `h4` and `p` text

```
time = event.find("h4").text.strip()
desc = event.find("p").text.strip()

print(f"{time}\t{desc}")
```

simple http server

- we need the `flask` package
- create a `flask` app
- define a `function` that will be our endpoint
- add a new `route` to this function
- start the server
- connect to the server through the browser and Python

simple http server

- we need the `flask` package
- create a `flask` app

```
app = Flask(__name__)
```

- define a `function` that will be our endpoint
- add a new `route` to this function
- start the server
- connect to the server through the browser and Python

simple http server

- we need the `flask` package
- create a `flask` app
- define a `function` that will be our endpoint
- add a new `route` to this function
- start the server
- connect to the server through the browser and Python

```
app = Flask(__name__)

def hello():
    return "Hello, World!"
```

simple http server

- we need the `flask` package

- create a `flask` app

```
app = Flask(__name__)
```

- define a `function` that will be our endpoint

```
def hello():
    return "Hello, World!"
```

- add a new `route` to this function

```
@app.route("/")
```

- start the server

- connect to the server through the browser and Python

simple http server

- we need the `flask` package

- create a `flask` app

```
app = Flask(__name__)
```

- define a `function` that will be our endpoint

```
def hello():
    return "Hello, World!"
```

- add a new `route` to this function

```
@app.route("/")
```

- start the server

```
FLASK_APP=examples/hello_server.py flask run
```

- connect to the server through the browser and Python

simple http server

- we need the `flask` package

- create a `flask` app

```
app = Flask(__name__)
```

- define a `function` that will be our endpoint

```
def hello():
    return "Hello, World!"
```

- add a new `route` to this function

```
@app.route("/")
```

- start the server

```
FLASK_APP=examples/hello_server.py flask run
```

- connect to the server through the browser and Python

```
requests.get('http://127.0.0.1:5000/?name=banana').text
```

wireshark log parser

- we need the `dpkt` package and `.pcap` file: http://tcpreplay.appneta.com/wiki/captures.html
- read a `.pcap` file
- create a `pdpkt.pcap.Reader` object
- iterate over packets and convert them to `Ethernet`
- extract packet's source and destinations IPs

wireshark log parser

- we need the `dpkt` package and `.pcap` file: http://tcpreplay.appneta.com/wiki/captures.html
- read a `.pcap` file

```
f = open(pcap_fname, "rb")
```

- create a `pdpkt.pcap.Reader` object
- iterate over packets and convert them to `Ethernet`
- extract packet's source and destinations IPs

wireshark log parser

- we need the `dpkt` package and `.pcap` file: http://tcpreplay.appneta.com/wiki/captures.html
- read a `.pcap` file

```
f = open(pcap_fname, "rb")
```

- create a `pdpkt.pcap.Reader` object

```
pcap = dpkt.pcap.Reader(f)
```

- iterate over packets and convert them to `Ethernet`
- extract packet's source and destinations IPs

wireshark log parser

- we need the `dpkt` package and `.pcap` file: http://tcpreplay.appneta.com/wiki/captures.html
- read a `.pcap` file

```
f = open(pcap_fname, "rb")
```

- create a `pdpkt.pcap.Reader` object

```
pcap = dpkt.pcap.Reader(f)
```

- iterate over packets and convert them to `Ethernet`

```
for ts, buf in pcap:
    eth = dpkt.ethernet.Ethernet(buf)
```

- extract packet's source and destinations IPs

wireshark log parser

- we need the `dpkt` package and `.pcap` file: http://tcpreplay.appneta.com/wiki/captures.html

- read a `.pcap` file

```
f = open(pcap_fname, "rb")
```

- create a `pdpkt.pcap.Reader` object

```
pcap = dpkt.pcap.Reader(f)
```

- iterate over packets and convert them to `Ethernet`

```
for ts, buf in pcap:
    eth = dpkt.ethernet.Ethernet(buf)
```

- extract packet's source and destinations IPs

```
if isinstance(eth.data, dpkt.ip.IP):
    ip = eth.data
    src = socket.inet_ntoa(ip.src)
    dst = socket.inet_ntoa(ip.dst)
```

other libraries and tools that could be handy

the data scientist new home

`jupyter`	`notebook`	write Python and tell a story
	`lab`	notebooks improved
	`server`	centralized notebooks/lab
`& google's colab!`		notebooks in the cloud

the wonderful world of (data) viz

`matplotlib`	basic library for plotting
`seaborn`	fancier plots
`bokeh`	fancier and interactive plots

scientific Python libraries

`numpy`	multi-dimensional arrays
`sklearn`	data mining and data analysis
`pandas`	data frames (i.e. tables) support

thanks for the attention

```
import antigravity
```