





*Variables are used to store values. A string is a series of characters, surrounded by single or double quotes.*

Hello world

print("Hello world!")

Hello world with a variable

msg = "Hello world!"

print(msg)

Concatenation (combining strings)

first\_name = 'albert'

last\_name = 'einstein'

full\_name = first\_name + ' ' + last\_name print(full\_name)



*A list stores a series of items in a particular order. You access items using an index, or within a loop.*

Make a list

bikes = ['trek', 'redline', 'giant'] Get the first item in a list

first\_bike = bikes[0]

Get the last item in a list

last\_bike = bikes[-1]

Looping through a list

for bike in bikes:

print(bike)

Adding items to a list

bikes = []

bikes.append('trek')

bikes.append('redline')

bikes.append('giant')

Making numerical lists

squares = []

for x in range(1, 11):

squares.append(x\*\*2)

List comprehensions

squares = [x\*\*2 for x in range(1, 11)] Slicing a list

finishers = ['sam', 'bob', 'ada', 'bea'] first\_two = finishers[:2]

Copying a list

copy\_of\_bikes = bikes[:]



*Tuples are similar to lists, but the items in a tuple can't be modified.*

Making a tuple

dimensions = (1920, 1080)



*If statements are used to test for particular conditions and respond appropriately.*

Conditional tests

equals x == 42

not equal x != 42

greater than x > 42

or equal to x >= 42

less than x < 42

or equal to x <= 42

Conditional test with lists

'trek' in bikes

'surly' not in bikes

Assigning boolean values

game\_active = True

can\_edit = False

A simple if test

if age >= 18:

print("You can vote!")

If-elif-else statements

if age < 4:

ticket\_price = 0

elif age < 18:

ticket\_price = 10

else:

ticket\_price = 15

*Dictionaries store connections between pieces of information. Each item in a dictionary is a key-value pair.*

A simple dictionary

alien = {'color': 'green', 'points': 5} Accessing a value

print("The alien's color is " + alien['color']) Adding a new key-value pair

alien['x\_position'] = 0

Looping through all key-value pairs

fav\_numbers = {'eric': 17, 'ever': 4} for name, number in fav\_numbers.items(): print(name + ' loves ' + str(number))

Looping through all keys

fav\_numbers = {'eric': 17, 'ever': 4} for name in fav\_numbers.keys():

print(name + ' loves a number')

Looping through all the values

fav\_numbers = {'eric': 17, 'ever': 4} for number in fav\_numbers.values():

print(str(number) + ' is a favorite')



*Your programs can prompt the user for input. All input is stored as a string.*

Prompting for a value

name = input("What's your name? ")

print("Hello, " + name + "!")

Prompting for numerical input

age = input("How old are you? ")

age = int(age)

pi = input("What's the value of pi? ") pi = float(pi)

*Covers Python 3 and Python 2 *

**

**

*A while loop repeats a block of code as long as a certain condition is true.*

A simple while loop

current\_value = 1

while current\_value <= 5:

print(current\_value)

current\_value += 1

Letting the user choose when to quit

msg = ''

while msg != 'quit':

msg = input("What's your message? ") print(msg)



*Functions are named blocks of code, designed to do one specific job. Information passed to a function is called an argument, and information received by a function is called a parameter.*

A simple function

def greet\_user():

"""Display a simple greeting."""

print("Hello!")

greet\_user()

Passing an argument

def greet\_user(username):

"""Display a personalized greeting.""" print("Hello, " + username + "!")

greet\_user('jesse')

Default values for parameters

def make\_pizza(topping='bacon'):

"""Make a single-topping pizza.""" print("Have a " + topping + " pizza!")

make\_pizza()

make\_pizza('pepperoni')

Returning a value

def add\_numbers(x, y):

"""Add two numbers and return the sum.""" return x + y

sum = add\_numbers(3, 5)

*A class defines the behavior of an object and the kind of information an object can store. The information in a class is stored in attributes, and functions that belong to a class are called methods. A child class inherits the attributes and methods from its parent class.*

Creating a dog class

class Dog():

"""Represent a dog."""

def \_\_init\_\_(self, name):

"""Initialize dog object."""

self.name = name

def sit(self):

"""Simulate sitting."""

print(self.name + " is sitting.") my\_dog = Dog('Peso')

print(my\_dog.name + " is a great dog!") my\_dog.sit()

Inheritance

class SARDog(Dog):

"""Represent a search dog."""

def \_\_init\_\_(self, name):

"""Initialize the sardog."""

super().\_\_init\_\_(name)

def search(self):

"""Simulate searching."""

print(self.name + " is searching.") my\_dog = SARDog('Willie')

print(my\_dog.name + " is a search dog.") my\_dog.sit()

my\_dog.search()



*If you had infinite programming skills, what would you build?*

As you're learning to program, it's helpful to think about the real-world projects you'd like to create. It's a good habit to keep an "ideas" notebook that you can refer to whenever you want to start a new project. If you haven't done so already, take a few minutes

*Your programs can read from files and write to files. Files are opened in read mode ('r') by default, but can also be opened in write mode ('w') and append mode ('a').*

Reading a file and storing its lines

filename = 'siddhartha.txt'

with open(filename) as file\_object:

lines = file\_object.readlines()

for line in lines:

print(line)

Writing to a file

filename = 'journal.txt'

with open(filename, 'w') as file\_object: file\_object.write("I love programming.")

Appending to a file

filename = 'journal.txt'

with open(filename, 'a') as file\_object: file\_object.write("\nI love making games.")



*Exceptions help you respond appropriately to errors that are likely to occur. You place code that might cause an error in the try block. Code that should run in response to an error goes in the except block. Code that should run only if the try block was successful goes in the else block.*

Catching an exception

prompt = "How many tickets do you need? " num\_tickets = input(prompt)

try:

num\_tickets = int(num\_tickets)

except ValueError:

print("Please try again.")

else:

print("Your tickets are printing.")



*Simple is better than complex*

If you have a choice between a simple and a complex solution, and both work, use the simple solution. Your code will be easier to maintain, and it will be easier for you and others to build on that code later on.

print(sum) *More cheat sheets available at*

and describe three projects you'd like to create.









A list stores a series of items in a particular order. Lists allow you to store sets of information in one place, whether you have just a few items or millions of items. Lists are one of Python's most powerful features readily accessible to new programmers, and they tie together many important concepts in programming.



*Use square brackets to define a list, and use commas to separate individual items in the list. Use plural names for lists, to make your code easier to read.*

Making a list

users = ['val', 'bob', 'mia', 'ron', 'ned']



*Individual elements in a list are accessed according to their position, called the index. The index of the first element is 0, the index of the second element is 1, and so forth. Negative indices refer to items at the end of the list. To get a particular element, write the name of the list and then the index of the element in square brackets.*

Getting the first element

first\_user = users[0]

Getting the second element

second\_user = users[1]

Getting the last element

newest\_user = users[-1]



*Once you've defined a list, you can change individual elements in the list. You do this by referring to the index of the item you want to modify.*

Changing an element

users[0] = 'valerie'

users[-2] = 'ronald'

*You can add elements to the end of a list, or you can insert them wherever you like in a list.*

Adding an element to the end of the list

users.append('amy')

Starting with an empty list

users = []

users.append('val')

users.append('bob')

users.append('mia')

Inserting elements at a particular position

users.insert(0, 'joe')

users.insert(3, 'bea')



*You can remove elements by their position in a list, or by the value of the item. If you remove an item by its value, Python removes only the first item that has that value.*

Deleting an element by its position

del users[-1]

Removing an item by its value

users.remove('mia')



*If you want to work with an element that you're removing from the list, you can "pop" the element. If you think of the list as a stack of items, pop() takes an item off the top of the stack. By default pop() returns the last element in the list, but you can also pop elements from any position in the list.*

Pop the last item from a list

most\_recent\_user = users.pop()

print(most\_recent\_user)

Pop the first item in a list

first\_user = users.pop(0)

print(first\_user)



*The len() function returns the number of items in a list.* Find the length of a list

num\_users = len(users)

print("We have " + str(num\_users) + " users.")

*The sort() method changes the order of a list permanently. The sorted() function returns a copy of the list, leaving the original list unchanged. You can sort the items in a list in alphabetical order, or reverse alphabetical order. You can also reverse the original order of the list. Keep in mind that lowercase and uppercase letters may affect the sort order.*

Sorting a list permanently

users.sort()

Sorting a list permanently in reverse alphabetical order

users.sort(reverse=True)

Sorting a list temporarily

print(sorted(users))

print(sorted(users, reverse=True))

Reversing the order of a list

users.reverse()



*Lists can contain millions of items, so Python provides an efficient way to loop through all the items in a list. When you set up a loop, Python pulls each item from the list one at a time and stores it in a temporary variable, which you provide a name for. This name should be the singular version of the list name.*

*The indented block of code makes up the body of the loop, where you can work with each individual item. Any lines that are not indented run after the loop is completed.*

Printing all items in a list

for user in users:

print(user)

Printing a message for each item, and a separate message afterwards

for user in users:

print("Welcome, " + user + "!")

print("Welcome, we're glad to see you all!") *Covers Python 3 and Python 2 *

**

**

*You can use the range() function to work with a set of numbers efficiently. The range() function starts at 0 by default, and stops one number below the number passed to it. You can use the list() function to efficiently generate a large list of numbers.*

Printing the numbers 0 to 1000

for number in range(1001):

print(number)

Printing the numbers 1 to 1000

for number in range(1, 1001):

print(number)

Making a list of numbers from 1 to a million numbers = list(range(1, 1000001))



*There are a number of simple statistics you can run on a list containing numerical data.*

Finding the minimum value in a list

ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77] youngest = min(ages)

Finding the maximum value

ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77] oldest = max(ages)

Finding the sum of all values

ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77] total\_years = sum(ages)



*You can work with any set of elements from a list. A portion of a list is called a slice. To slice a list start with the index of the first item you want, then add a colon and the index after the last item you want. Leave off the first index to start at*

*the beginning of the list, and leave off the last index to slice through the end of the list.*

Getting the first three items

finishers = ['kai', 'abe', 'ada', 'gus', 'zoe'] first\_three = finishers[:3]

Getting the middle three items

middle\_three = finishers[1:4]

Getting the last three items

last\_three = finishers[-3:]

*To copy a list make a slice that starts at the first item and ends at the last item. If you try to copy a list without using this approach, whatever you do to the copied list will affect the original list as well.*

Making a copy of a list

finishers = ['kai', 'abe', 'ada', 'gus', 'zoe'] copy\_of\_finishers = finishers[:]



*You can use a loop to generate a list based on a range of numbers or on another list. This is a common operation, so Python offers a more efficient way to do it. List comprehensions may look complicated at first; if so, use the for loop approach until you're ready to start using comprehensions.*

*To write a comprehension, define an expression for the values you want to store in the list. Then write a for loop to generate input values needed to make the list.*

Using a loop to generate a list of square numbers

squares = []

for x in range(1, 11):

square = x\*\*2

squares.append(square)

Using a comprehension to generate a list of square numbers

squares = [x\*\*2 for x in range(1, 11)] Using a loop to convert a list of names to upper case names = ['kai', 'abe', 'ada', 'gus', 'zoe']

upper\_names = []

for name in names:

upper\_names.append(name.upper())

Using a comprehension to convert a list of names to upper case

names = ['kai', 'abe', 'ada', 'gus', 'zoe'] upper\_names = [name.upper() for name in names]



*Readability counts*

∙ Use four spaces per indentation level.

∙ Keep your lines to 79 characters or fewer. ∙ Use single blank lines to group parts of your program visually.

*A tuple is like a list, except you can't change the values in a tuple once it's defined. Tuples are good for storing information that shouldn't be changed throughout the life of a program. Tuples are designated by parentheses instead of square brackets. (You can overwrite an entire tuple, but you can't change the individual elements in a tuple.)*

Defining a tuple

dimensions = (800, 600)

Looping through a tuple

for dimension in dimensions:

print(dimension)

Overwriting a tuple

dimensions = (800, 600)

print(dimensions)

dimensions = (1200, 900)



*When you're first learning about data structures such as lists, it helps to visualize how Python is working with the information in your program. pythontutor.com is a great tool for seeing how Python keeps track of the information in a list. Try running the following code on pythontutor.com, and then run your own code.*

Build a list and print the items in the list

dogs = []

dogs.append('willie')

dogs.append('hootz')

dogs.append('peso')

dogs.append('goblin')

for dog in dogs:

print("Hello " + dog + "!")

print("I love these dogs!")

print("\nThese were my first two dogs:") old\_dogs = dogs[:2]

for old\_dog in old\_dogs:

print(old\_dog)

del dogs[0]

dogs.remove('peso')

print(dogs)

*More cheat sheets available at*

**

**

**

**

Python's dictionaries allow you to connect pieces of related information. Each piece of information in a dictionary is stored as a key-value pair. When you provide a key, Python returns the value associated with that key. You can loop through all the key-value pairs, all the keys, or all the values.



*Use curly braces to define a dictionary. Use colons to connect keys and values, and use commas to separate individual key-value pairs.*

Making a dictionary

alien\_0 = {'color': 'green', 'points': 5}



*To access the value associated with an individual key give the name of the dictionary and then place the key in a set of square brackets. If the key you're asking for is not in the dictionary, an error will occur.*

*You can also use the get() method, which returns None instead of an error if the key doesn't exist. You can also specify a default value to use if the key is not in the dictionary.*

Getting the value associated with a key

alien\_0 = {'color': 'green', 'points': 5}

print(alien\_0['color'])

print(alien\_0['points'])

Getting the value with get()

alien\_0 = {'color': 'green'}

alien\_color = alien\_0.get('color')

alien\_points = alien\_0.get('points', 0)

print(alien\_color)

print(alien\_points)

*You can store as many key-value pairs as you want in a dictionary, until your computer runs out of memory. To add a new key-value pair to an existing dictionary give the name of the dictionary and the new key in square brackets, and set it equal to the new value.*

*This also allows you to start with an empty dictionary and add key-value pairs as they become relevant.*

Adding a key-value pair

alien\_0 = {'color': 'green', 'points': 5}

alien\_0['x'] = 0

alien\_0['y'] = 25

alien\_0['speed'] = 1.5

Adding to an empty dictionary

alien\_0 = {}

alien\_0['color'] = 'green'

alien\_0['points'] = 5



*You can modify the value associated with any key in a dictionary. To do so give the name of the dictionary and enclose the key in square brackets, then provide the new value for that key.*

Modifying values in a dictionary

alien\_0 = {'color': 'green', 'points': 5} print(alien\_0)

# Change the alien's color and point value. alien\_0['color'] = 'yellow'

alien\_0['points'] = 10

print(alien\_0)



*You can remove any key-value pair you want from a dictionary. To do so use the del keyword and the dictionary name, followed by the key in square brackets. This will delete the key and its associated value.*

Deleting a key-value pair

alien\_0 = {'color': 'green', 'points': 5} print(alien\_0)

del alien\_0['points']

print(alien\_0)



*Try running some of these examples on pythontutor.com.*

*You can loop through a dictionary in three ways: you can loop through all the key-value pairs, all the keys, or all the values.*

*A dictionary only tracks the connections between keys and values; it doesn't track the order of items in the dictionary. If you want to process the information in order, you can sort the keys in your loop.*

Looping through all key-value pairs

# Store people's favorite languages. fav\_languages = {

'jen': 'python',

'sarah': 'c',

'edward': 'ruby',

'phil': 'python',

}

# Show each person's favorite language. for name, language in fav\_languages.items(): print(name + ": " + language)

Looping through all the keys

# Show everyone who's taken the survey. for name in fav\_languages.keys():

print(name)

Looping through all the values

# Show all the languages that have been chosen. for language in fav\_languages.values(): print(language)

Looping through all the keys in order

# Show each person's favorite language, # in order by the person's name.

for name in sorted(fav\_languages.keys()): print(name + ": " + language)



*You can find the number of key-value pairs in a dictionary.* Finding a dictionary's length

num\_responses = len(fav\_languages)

*Covers Python 3 and Python 2 *

**

**

*It's sometimes useful to store a set of dictionaries in a list; this is called nesting.*

Storing dictionaries in a list

# Start with an empty list.

users = []

# Make a new user, and add them to the list. new\_user = {

'last': 'fermi',

'first': 'enrico',

'username': 'efermi',

}

users.append(new\_user)

# Make another new user, and add them as well. new\_user = {

'last': 'curie',

'first': 'marie',

'username': 'mcurie',

}

users.append(new\_user)

# Show all information about each user. for user\_dict in users:

for k, v in user\_dict.items():

print(k + ": " + v)

print("\n")

You can also define a list of dictionaries directly, without using append():

# Define a list of users, where each user # is represented by a dictionary.

users = [

{

'last': 'fermi',

'first': 'enrico',

'username': 'efermi',

},

{

'last': 'curie',

'first': 'marie',

'username': 'mcurie',

},

]

# Show all information about each user. for user\_dict in users:

for k, v in user\_dict.items():

print(k + ": " + v)

print("\n")

*Storing a list inside a dictionary alows you to associate more than one value with each key.*

Storing lists in a dictionary

# Store multiple languages for each person. fav\_languages = {

'jen': ['python', 'ruby'],

'sarah': ['c'],

'edward': ['ruby', 'go'],

'phil': ['python', 'haskell'],

}

# Show all responses for each person. for name, langs in fav\_languages.items(): print(name + ": ")

for lang in langs:

print("- " + lang)

*You can store a dictionary inside another dictionary. In this case each value associated with a key is itself a dictionary.*

Storing dictionaries in a dictionary

users = {

'aeinstein': {

'first': 'albert',

'last': 'einstein',

'location': 'princeton',

},

'mcurie': {

'first': 'marie',

'last': 'curie',

'location': 'paris',

},

}

for username, user\_dict in users.items(): print("\nUsername: " + username) full\_name = user\_dict['first'] + " " full\_name += user\_dict['last']

location = user\_dict['location']

print("\tFull name: " + full\_name.title()) print("\tLocation: " + location.title())



*Nesting is extremely useful in certain situations. However, be aware of making your code overly complex. If you're nesting items much deeper than what you see here there are probably simpler ways of managing your data, such as using classes.*

*Standard Python dictionaries don't keep track of the order in which keys and values are added; they only preserve the association between each key and its value. If you want to preserve the order in which keys and values are added, use an OrderedDict.*

Preserving the order of keys and values

from collections import OrderedDict

# Store each person's languages, keeping # track of who respoded first.

fav\_languages = OrderedDict()

fav\_languages['jen'] = ['python', 'ruby'] fav\_languages['sarah'] = ['c']

fav\_languages['edward'] = ['ruby', 'go'] fav\_languages['phil'] = ['python', 'haskell']

# Display the results, in the same order they # were entered.

for name, langs in fav\_languages.items(): print(name + ":")

for lang in langs:

print("- " + lang)



*You can use a loop to generate a large number of dictionaries efficiently, if all the dictionaries start out with similar data.*

A million aliens

aliens = []

# Make a million green aliens, worth 5 points # each. Have them all start in one row. for alien\_num in range(1000000):

new\_alien = {}

new\_alien['color'] = 'green'

new\_alien['points'] = 5

new\_alien['x'] = 20 \* alien\_num

new\_alien['y'] = 0

aliens.append(new\_alien)

# Prove the list contains a million aliens. num\_aliens = len(aliens)

print("Number of aliens created:")

print(num\_aliens)

*More cheat sheets available at*

**

**

**

**

**

If statements allow you to examine the current state of a program and respond appropriately to that state. You can write a simple if statement that checks one condition, or you can create a complex series of if statements that idenitfy the exact conditions you're looking for.

While loops run as long as certain conditions remain true. You can use while loops to let your programs run as long as your users want them to.



*A conditional test is an expression that can be evaluated as True or False. Python uses the values True and False to decide whether the code in an if statement should be executed.*

Checking for equality

*A single equal sign assigns a value to a variable. A double equal sign (==) checks whether two values are equal.*

>>> car = 'bmw'

>>> car == 'bmw'

True

>>> car = 'audi'

>>> car == 'bmw'

False

Ignoring case when making a comparison

>>> car = 'Audi'

>>> car.lower() == 'audi'

True

Checking for inequality

>>> topping = 'mushrooms'

>>> topping != 'anchovies'

True

*Testing numerical values is similar to testing string values.* Testing equality and inequality

>>> age = 18

>>> age == 18

True

>>> age != 18

False

Comparison operators

>>> age = 19

>>> age < 21

True

>>> age <= 21

True

>>> age > 21

False

>>> age >= 21

False



*You can check multiple conditions at the same time. The and operator returns True if all the conditions listed are True. The or operator returns True if any condition is True.*

Using and to check multiple conditions

>>> age\_0 = 22

>>> age\_1 = 18

>>> age\_0 >= 21 and age\_1 >= 21

False

>>> age\_1 = 23

>>> age\_0 >= 21 and age\_1 >= 21

True

Using or to check multiple conditions

>>> age\_0 = 22

>>> age\_1 = 18

>>> age\_0 >= 21 or age\_1 >= 21

True

>>> age\_0 = 18

>>> age\_0 >= 21 or age\_1 >= 21

False



*A boolean value is either True or False. Variables with boolean values are often used to keep track of certain conditions within a program.*

Simple boolean values

game\_active = True

can\_edit = False

*Several kinds of if statements exist. Your choice of which to use depends on the number of conditions you need to test. You can have as many elif blocks as you need, and the else block is always optional.*

Simple if statement

age = 19

if age >= 18:

print("You're old enough to vote!") If-else statements

age = 17

if age >= 18:

print("You're old enough to vote!") else:

print("You can't vote yet.")

The if-elif-else chain

age = 12

if age < 4:

price = 0

elif age < 18:

price = 5

else:

price = 10

print("Your cost is $" + str(price) + ".")



*You can easily test whether a certain value is in a list. You can also test whether a list is empty before trying to loop through the list.*

Testing if a value is in a list

>>> players = ['al', 'bea', 'cyn', 'dale'] >>> 'al' in players

True

>>> 'eric' in players

False

*Covers Python 3 and Python 2 *

**

**

**

Testing if a value is not in a list

banned\_users = ['ann', 'chad', 'dee'] user = 'erin'

if user not in banned\_users:

print("You can play!")

Checking if a list is empty

players = []

if players:

for player in players:

print("Player: " + player.title()) else:

print("We have no players yet!")



*You can allow your users to enter input using the input() statement. In Python 3, all input is stored as a string.*

Simple input

name = input("What's your name? ")

print("Hello, " + name + ".")

Accepting numerical input

age = input("How old are you? ")

age = int(age)

if age >= 18:

print("\nYou can vote!")

else:

print("\nYou can't vote yet.")

Accepting input in Python 2.7

*Use raw\_input() in Python 2.7. This function interprets all input as a string, just as input() does in Python 3.*

name = raw\_input("What's your name? ") print("Hello, " + name + ".")



*A while loop repeats a block of code as long as a condition is True.*

Counting to 5

Letting the user choose when to quit

prompt = "\nTell me something, and I'll " prompt += "repeat it back to you."

prompt += "\nEnter 'quit' to end the program. "

message = ""

while message != 'quit':

message = input(prompt)

if message != 'quit':

print(message)

Using a flag

prompt = "\nTell me something, and I'll " prompt += "repeat it back to you."

prompt += "\nEnter 'quit' to end the program. "

active = True

while active:

message = input(prompt)

if message == 'quit':

active = False

else:

print(message)

Using break to exit a loop

prompt = "\nWhat cities have you visited?" prompt += "\nEnter 'quit' when you're done. "

while True:

city = input(prompt)

if city == 'quit':

break

else:

print("I've been to " + city + "!")



*Sublime Text doesn't run programs that prompt the user for input. You can use Sublime Text to write programs that prompt for input, but you'll need to run these programs from a terminal.*

**

Using continue in a loop

banned\_users = ['eve', 'fred', 'gary', 'helen']

prompt = "\nAdd a player to your team." prompt += "\nEnter 'quit' when you're done. "

players = []

while True:

player = input(prompt)

if player == 'quit':

break

elif player in banned\_users:

print(player + " is banned!") continue

else:

players.append(player)

print("\nYour team:")

for player in players:

print(player)



*Every while loop needs a way to stop running so it won't continue to run forever. If there's no way for the condition to become False, the loop will never stop running.*

An infinite loop

while True:

name = input("\nWho are you? ")

print("Nice to meet you, " + name + "!")

*The remove() method removes a specific value from a list, but it only removes the first instance of the value you provide. You can use a while loop to remove all instances of a particular value.*

Removing all cats from a list of pets

pets = ['dog', 'cat', 'dog', 'fish', 'cat', 'rabbit', 'cat']

print(pets)

while 'cat' in pets:

pets.remove('cat')

current\_number = 1

while current\_number <= 5:

*You can use the break statement and the continue* print(pets) *statement with any of Python's loops. For example you can*

print(current\_number) current\_number += 1

*use break to quit a for loop that's working through a list or a dictionary. You can use continue to skip over certain items when looping through a list or dictionary as well.*

*More cheat sheets available at *

**

**

**

Functions are named blocks of code designed to do one specific job. Functions allow you to write code once that can then be run whenever you need to accomplish the same task. Functions can take in the information they need, and return the information they generate. Using functions effectively makes your programs easier to write, read, test, and fix.



*The first line of a function is its definition, marked by the keyword def. The name of the function is followed by a set of parentheses and a colon. A docstring, in triple quotes, describes what the function does. The body of a function is indented one level.*

*To call a function, give the name of the function followed by a set of parentheses.*

Making a function

def greet\_user():

"""Display a simple greeting."""

print("Hello!")

greet\_user()



*Information that's passed to a function is called an argument; information that's received by a function is called a parameter. Arguments are included in parentheses after the function's name, and parameters are listed in parentheses in the function's definition.*

Passing a single argument

def greet\_user(username):

"""Display a simple greeting."""

print("Hello, " + username + "!")

greet\_user('jesse')

greet\_user('diana')

greet\_user('brandon')

*The two main kinds of arguments are positional and keyword arguments. When you use positional arguments Python matches the first argument in the function call with the first parameter in the function definition, and so forth. With keyword arguments, you specify which parameter each argument should be assigned to in the function call. When you use keyword arguments, the order of the arguments doesn't matter.*

Using positional arguments

def describe\_pet(animal, name):

"""Display information about a pet.""" print("\nI have a " + animal + ".") print("Its name is " + name + ".")

describe\_pet('hamster', 'harry')

describe\_pet('dog', 'willie')

Using keyword arguments

def describe\_pet(animal, name):

"""Display information about a pet.""" print("\nI have a " + animal + ".") print("Its name is " + name + ".")

describe\_pet(animal='hamster', name='harry') describe\_pet(name='willie', animal='dog')



*You can provide a default value for a parameter. When function calls omit this argument the default value will be used. Parameters with default values must be listed after parameters without default values in the function's definition so positional arguments can still work correctly.*

Using a default value

def describe\_pet(name, animal='dog'): """Display information about a pet.""" print("\nI have a " + animal + ".") print("Its name is " + name + ".")

describe\_pet('harry', 'hamster')

describe\_pet('willie')

Using None to make an argument optional

def describe\_pet(animal, name=None):

"""Display information about a pet.""" print("\nI have a " + animal + ".") if name:

print("Its name is " + name + ".")

describe\_pet('hamster', 'harry')

describe\_pet('snake')

*A function can return a value or a set of values. When a function returns a value, the calling line must provide a variable in which to store the return value. A function stops running when it reaches a return statement.*

Returning a single value

def get\_full\_name(first, last):

"""Return a neatly formatted full name.""" full\_name = first + ' ' + last

return full\_name.title()

musician = get\_full\_name('jimi', 'hendrix') print(musician)

Returning a dictionary

def build\_person(first, last):

"""Return a dictionary of information about a person.

"""

person = {'first': first, 'last': last} return person

musician = build\_person('jimi', 'hendrix') print(musician)

Returning a dictionary with optional values

def build\_person(first, last, age=None): """Return a dictionary of information about a person.

"""

person = {'first': first, 'last': last} if age:

person['age'] = age

return person

musician = build\_person('jimi', 'hendrix', 27) print(musician)

musician = build\_person('janis', 'joplin') print(musician)



*Try running some of these examples on pythontutor.com.Covers Python 3 and Python 2 *

**

**

*You can pass a list as an argument to a function, and the function can work with the values in the list. Any changes the function makes to the list will affect the original list. You can prevent a function from modifying a list by passing a copy of the list as an argument.*

Passing a list as an argument

def greet\_users(names):

"""Print a simple greeting to everyone.""" for name in names:

msg = "Hello, " + name + "!"

print(msg)

usernames = ['hannah', 'ty', 'margot'] greet\_users(usernames)

Allowing a function to modify a list

*The following example sends a list of models to a function for printing. The original list is emptied, and the second list is filled.*

def print\_models(unprinted, printed): """3d print a set of models."""

while unprinted:

current\_model = unprinted.pop() print("Printing " + current\_model) printed.append(current\_model)

# Store some unprinted designs,

# and print each of them.

unprinted = ['phone case', 'pendant', 'ring'] printed = []

print\_models(unprinted, printed)

print("\nUnprinted:", unprinted)

print("Printed:", printed)

Preventing a function from modifying a list *The following example is the same as the previous one, except the original list is unchanged after calling print\_models().*

def print\_models(unprinted, printed): """3d print a set of models."""

while unprinted:

current\_model = unprinted.pop() print("Printing " + current\_model) printed.append(current\_model)

# Store some unprinted designs,

# and print each of them.

original = ['phone case', 'pendant', 'ring'] printed = []

print\_models(original[:], printed)

print("\nOriginal:", original)

print("Printed:", printed)

*Sometimes you won't know how many arguments a function will need to accept. Python allows you to collect an arbitrary number of arguments into one parameter using the \* operator. A parameter that accepts an arbitrary number of arguments must come last in the function definition. The \*\* operator allows a parameter to collect an arbitrary number of keyword arguments.*

Collecting an arbitrary number of arguments

def make\_pizza(size, \*toppings):

"""Make a pizza."""

print("\nMaking a " + size + " pizza.") print("Toppings:")

for topping in toppings:

print("- " + topping)

# Make three pizzas with different toppings. make\_pizza('small', 'pepperoni')

make\_pizza('large', 'bacon bits', 'pineapple') make\_pizza('medium', 'mushrooms', 'peppers', 'onions', 'extra cheese')

Collecting an arbitrary number of keyword arguments

def build\_profile(first, last, \*\*user\_info): """Build a user's profile dictionary.""" # Build a dict with the required keys. profile = {'first': first, 'last': last}

# Add any other keys and values.

for key, value in user\_info.items(): profile[key] = value

return profile

# Create two users with different kinds # of information.

user\_0 = build\_profile('albert', 'einstein', location='princeton')

user\_1 = build\_profile('marie', 'curie', location='paris', field='chemistry')

print(user\_0)

print(user\_1)

*As you can see there are many ways to write and call a function. When you're starting out, aim for something that simply works. As you gain experience you'll develop an understanding of the more subtle advantages of different structures such as positional and keyword arguments, and the various approaches to importing functions. For now if your functions do what you need them to, you're doing well.*

*You can store your functions in a separate file called a module, and then import the functions you need into the file containing your main program. This allows for cleaner program files. (Make sure your module is stored in the same directory as your main program.)*

Storing a function in a module

*File: pizza.py*

def make\_pizza(size, \*toppings):

"""Make a pizza."""

print("\nMaking a " + size + " pizza.") print("Toppings:")

for topping in toppings:

print("- " + topping)

Importing an entire module

*File: making\_pizzas.py*

*Every function in the module is available in the program file.* import pizza

pizza.make\_pizza('medium', 'pepperoni') pizza.make\_pizza('small', 'bacon', 'pineapple')

Importing a specific function

*Only the imported functions are available in the program file.* from pizza import make\_pizza

make\_pizza('medium', 'pepperoni')

make\_pizza('small', 'bacon', 'pineapple') Giving a module an alias

import pizza as p

p.make\_pizza('medium', 'pepperoni')

p.make\_pizza('small', 'bacon', 'pineapple') Giving a function an alias

from pizza import make\_pizza as mp

mp('medium', 'pepperoni')

mp('small', 'bacon', 'pineapple')

Importing all functions from a module

*Don't do this, but recognize it when you see it in others' code. It can result in naming conflicts, which can cause errors.*

from pizza import \*

make\_pizza('medium', 'pepperoni')

make\_pizza('small', 'bacon', 'pineapple')

*More cheat sheets available at*

**

**

**

**

Classes are the foundation of object-oriented programming. Classes represent real-world things you want to model in your programs: for example dogs, cars, and robots. You use a class to make objects, which are specific instances of dogs, cars, and robots. A class defines the general behavior that a whole category of objects can have, and the information that can be associated with those objects. Classes can inherit from each other – you can write a class that extends the functionality of an existing class. This allows you to code efficiently for a wide variety of situations.



*Consider how we might model a car. What information would we associate with a car, and what behavior would it have? The information is stored in variables called attributes, and the behavior is represented by functions. Functions that are part of a class are called methods.*

The Car class

class Car():

"""A simple attempt to model a car."""

def \_\_init\_\_(self, make, model, year): """Initialize car attributes.""" self.make = make

self.model = model

self.year = year

# Fuel capacity and level in gallons. self.fuel\_capacity = 15

self.fuel\_level = 0

def fill\_tank(self):

"""Fill gas tank to capacity.""" self.fuel\_level = self.fuel\_capacity print("Fuel tank is full.")

def drive(self):

"""Simulate driving."""

print("The car is moving.")

Creating an object from a class

my\_car = Car('audi', 'a4', 2016)

Accessing attribute values

print(my\_car.make)

print(my\_car.model)

print(my\_car.year)

Calling methods

my\_car.fill\_tank()

my\_car.drive()

Creating multiple objects

my\_car = Car('audi', 'a4', 2016)

my\_old\_car = Car('subaru', 'outback', 2013) my\_truck = Car('toyota', 'tacoma', 2010)



*You can modify an attribute's value directly, or you can write methods that manage updating values more carefully.*

Modifying an attribute directly

my\_new\_car = Car('audi', 'a4', 2016)

my\_new\_car.fuel\_level = 5

Writing a method to update an attribute's value

def update\_fuel\_level(self, new\_level): """Update the fuel level."""

if new\_level <= self.fuel\_capacity: self.fuel\_level = new\_level

else:

print("The tank can't hold that much!") Writing a method to increment an attribute's value

def add\_fuel(self, amount):

"""Add fuel to the tank."""

if (self.fuel\_level + amount

<= self.fuel\_capacity):

self.fuel\_level += amount

print("Added fuel.")

else:

print("The tank won't hold that much.")



*In Python class names are written in CamelCase and object names are written in lowercase with underscores. Modules that contain classes should still be named in lowercase with underscores.*

*If the class you're writing is a specialized version of another class, you can use inheritance. When one class inherits from another, it automatically takes on all the attributes and methods of the parent class. The child class is free to introduce new attributes and methods, and override attributes and methods of the parent class.*

*To inherit from another class include the name of the parent class in parentheses when defining the new class.*

The \_\_init\_\_() method for a child class

class ElectricCar(Car):

"""A simple model of an electric car."""

def \_\_init\_\_(self, make, model, year): """Initialize an electric car.""" super().\_\_init\_\_(make, model, year)

# Attributes specific to electric cars. # Battery capacity in kWh.

self.battery\_size = 70

# Charge level in %.

self.charge\_level = 0

Adding new methods to the child class

class ElectricCar(Car):

--snip--

def charge(self):

"""Fully charge the vehicle.""" self.charge\_level = 100

print("The vehicle is fully charged.") Using child methods and parent methods my\_ecar = ElectricCar('tesla', 'model s', 2016)

my\_ecar.charge()

my\_ecar.drive()



*There are many ways to model real world objects and situations in code, and sometimes that variety can feel overwhelming. Pick an approach and try it – if your first attempt doesn't work, try a different approach.*

*Covers Python 3 and Python 2 *

**

**

**

Overriding parent methods

class ElectricCar(Car):

--snip--

def fill\_tank(self):

"""Display an error message.""" print("This car has no fuel tank!")



*A class can have objects as attributes. This allows classes to work together to model complex situations.*

A Battery class

class Battery():

"""A battery for an electric car."""

def \_\_init\_\_(self, size=70):

"""Initialize battery attributes.""" # Capacity in kWh, charge level in %. self.size = size

self.charge\_level = 0

def get\_range(self):

"""Return the battery's range.""" if self.size == 70:

return 240

elif self.size == 85:

return 270

Using an instance as an attribute

class ElectricCar(Car):

--snip--

def \_\_init\_\_(self, make, model, year): """Initialize an electric car.""" super().\_\_init\_\_(make, model, year)

# Attribute specific to electric cars. self.battery = Battery()

def charge(self):

"""Fully charge the vehicle.""" self.battery.charge\_level = 100 print("The vehicle is fully charged.")

Using the instance

my\_ecar = ElectricCar('tesla', 'model x', 2016)

my\_ecar.charge()

print(my\_ecar.battery.get\_range())

my\_ecar.drive()

*Class files can get long as you add detailed information and functionality. To help keep your program files uncluttered, you can store your classes in modules and import the classes you need into your main program.*

Storing classes in a file

*car.py*

"""Represent gas and electric cars."""

class Car():

"""A simple attempt to model a car.""" --snip—

class Battery():

"""A battery for an electric car.""" --snip--

class ElectricCar(Car):

"""A simple model of an electric car.""" --snip--

Importing individual classes from a module *my\_cars.py*

from car import Car, ElectricCar

my\_beetle = Car('volkswagen', 'beetle', 2016) my\_beetle.fill\_tank()

my\_beetle.drive()

my\_tesla = ElectricCar('tesla', 'model s', 2016)

my\_tesla.charge()

my\_tesla.drive()

Importing an entire module

import car

my\_beetle = car.Car(

'volkswagen', 'beetle', 2016) my\_beetle.fill\_tank()

my\_beetle.drive()

my\_tesla = car.ElectricCar(

'tesla', 'model s', 2016)

my\_tesla.charge()

my\_tesla.drive()

Importing all classes from a module

*(Don’t do this, but recognize it when you see it.)*

from car import \*

my\_beetle = Car('volkswagen', 'beetle', 2016)

Classes should inherit from object

class ClassName(object):

The Car class in Python 2.7

class Car(object):

Child class \_\_init\_\_() method is different

class ChildClassName(ParentClass):

def \_\_init\_\_(self):

super(ClassName, self).\_\_init\_\_() The ElectricCar class in Python 2.7

class ElectricCar(Car):

def \_\_init\_\_(self, make, model, year): super(ElectricCar, self).\_\_init\_\_( make, model, year)



*A list can hold as many items as you want, so you can make a large number of objects from a class and store them in a list.*

*Here's an example showing how to make a fleet of rental cars, and make sure all the cars are ready to drive.*

A fleet of rental cars

from car import Car, ElectricCar

# Make lists to hold a fleet of cars. gas\_fleet = []

electric\_fleet = []

# Make 500 gas cars and 250 electric cars. for \_ in range(500):

car = Car('ford', 'focus', 2016)

gas\_fleet.append(car)

for \_ in range(250):

ecar = ElectricCar('nissan', 'leaf', 2016) electric\_fleet.append(ecar)

# Fill the gas cars, and charge electric cars. for car in gas\_fleet:

car.fill\_tank()

for ecar in electric\_fleet:

ecar.charge()

print("Gas cars:", len(gas\_fleet))

print("Electric cars:", len(electric\_fleet))

*More cheat sheets available at*

**

**

**

Your programs can read information in from files, and they can write data to files. Reading from files allows you to work with a wide variety of information; writing to files allows users to pick up where they left off the

next time they run your program. You can write text to files, and you can store Python structures such as lists in data files.

Exceptions are special objects that help your programs respond to errors in appropriate ways. For example if your program tries to open a file that doesn’t exist, you can use exceptions to display an informative error message instead of having the program crash.



*To read from a file your program needs to open the file and then read the contents of the file. You can read the entire contents of the file at once, or read the file line by line. The with statement makes sure the file is closed properly when the program has finished accessing the file.*

Reading an entire file at once

filename = 'siddhartha.txt'

with open(filename) as f\_obj:

contents = f\_obj.read()

print(contents)

Reading line by line

*Each line that's read from the file has a newline character at the end of the line, and the print function adds its own newline character. The rstrip() method gets rid of the the extra blank lines this would result in when printing to the terminal.*

filename = 'siddhartha.txt'

with open(filename) as f\_obj:

for line in f\_obj:

print(line.rstrip())

Storing the lines in a list

filename = 'siddhartha.txt'

with open(filename) as f\_obj:

lines = f\_obj.readlines()

for line in lines:

print(line.rstrip())



*Passing the 'w' argument to open() tells Python you want to write to the file. Be careful; this will erase the contents of the file if it already exists. Passing the 'a' argument tells Python you want to append to the end of an existing file.*

Writing to an empty file

filename = 'programming.txt'

with open(filename, 'w') as f:

f.write("I love programming!")

Writing multiple lines to an empty file

filename = 'programming.txt'

with open(filename, 'w') as f:

f.write("I love programming!\n")

f.write("I love creating new games.\n") Appending to a file

filename = 'programming.txt'

with open(filename, 'a') as f:

f.write("I also love working with data.\n") f.write("I love making apps as well.\n")



*When Python runs the open() function, it looks for the file in the same directory where the program that's being excuted is stored. You can open a file from a subfolder using a relative path. You can also use an absolute path to open any file on your system.*

Opening a file from a subfolder

f\_path = "text\_files/alice.txt"

with open(f\_path) as f\_obj:

lines = f\_obj.readlines()

for line in lines:

print(line.rstrip())

Opening a file using an absolute path

f\_path = "/home/ehmatthes/books/alice.txt"

with open(f\_path) as f\_obj:

lines = f\_obj.readlines()

Opening a file on Windows

*Windows will sometimes interpret forward slashes incorrectly. If you run into this, use backslashes in your file paths.*

f\_path = "C:\Users\ehmatthes\books\alice.txt"

with open(f\_path) as f\_obj:

lines = f\_obj.readlines()



*When you think an error may occur, you can write a try except block to handle the exception that might be raised. The try block tells Python to try running some code, and the except block tells Python what to do if the code results in a particular kind of error.*

Handling the ZeroDivisionError exception

try:

print(5/0)

except ZeroDivisionError:

print("You can't divide by zero!") Handling the FileNotFoundError exception f\_name = 'siddhartha.txt'

try:

with open(f\_name) as f\_obj:

lines = f\_obj.readlines()

except FileNotFoundError:

msg = "Can't find file {0}.".format(f\_name) print(msg)

*It can be hard to know what kind of exception to handle when writing code. Try writing your code without a try block, and make it generate an error. The traceback will tell you what kind of exception your program needs to handle.*

*Covers Python 3 and Python 2 *

**

**

**

*The try block should only contain code that may cause an error. Any code that depends on the try block running successfully should be placed in the else block.*

Using an else block

print("Enter two numbers. I'll divide them.")

x = input("First number: ")

y = input("Second number: ")

try:

result = int(x) / int(y)

except ZeroDivisionError:

print("You can't divide by zero!") else:

print(result)

Preventing crashes from user input

*Without the except block in the following example, the program would crash if the user tries to divide by zero. As written, it will handle the error gracefully and keep running.*

"""A simple calculator for division only."""

print("Enter two numbers. I'll divide them.") print("Enter 'q' to quit.")

while True:

x = input("\nFirst number: ")

if x == 'q':

break

y = input("Second number: ")

if y == 'q':

break

try:

result = int(x) / int(y)

except ZeroDivisionError:

print("You can't divide by zero!") else:

print(result)



*Well-written, properly tested code is not very prone to internal errors such as syntax or logical errors. But every time your program depends on something external such as user input or the existence of a file, there's a possibility of an exception being raised.*

*It's up to you how to communicate errors to your users. Sometimes users need to know if a file is missing; sometimes it's better to handle the error silently. A little experience will help you know how much to report.*

*Sometimes you want your program to just continue running when it encounters an error, without reporting the error to the user. Using the pass statement in an else block allows you to do this.*

Using the pass statement in an else block

f\_names = ['alice.txt', 'siddhartha.txt', 'moby\_dick.txt', 'little\_women.txt']

for f\_name in f\_names:

# Report the length of each file found. try:

with open(f\_name) as f\_obj:

lines = f\_obj.readlines() except FileNotFoundError:

# Just move on to the next file. pass

else:

num\_lines = len(lines)

msg = "{0} has {1} lines.".format( f\_name, num\_lines)

print(msg)



*Exception-handling code should catch specific exceptions that you expect to happen during your program's execution. A bare except block will catch all exceptions, including keyboard interrupts and system exits you might need when forcing a program to close.*

*If you want to use a try block and you're not sure which exception to catch, use Exception. It will catch most exceptions, but still allow you to interrupt programs intentionally.*

Don’t use bare except blocks

try:

# Do something

except:

pass

Use Exception instead

try:

# Do something

except Exception:

pass

Printing the exception

try:

# Do something

except Exception as e:

print(e, type(e))

*The json module allows you to dump simple Python data structures into a file, and load the data from that file the next time the program runs. The JSON data format is not specific to Python, so you can share this kind of data with people who work in other languages as well.*

*Knowing how to manage exceptions is important when working with stored data. You'll usually want to make sure the data you're trying to load exists before working with it.*

Using json.dump() to store data

"""Store some numbers."""

import json

numbers = [2, 3, 5, 7, 11, 13]

filename = 'numbers.json'

with open(filename, 'w') as f\_obj:

json.dump(numbers, f\_obj)

Using json.load() to read data

"""Load some previously stored numbers.""" import json

filename = 'numbers.json'

with open(filename) as f\_obj:

numbers = json.load(f\_obj)

print(numbers)

Making sure the stored data exists

import json

f\_name = 'numbers.json'

try:

with open(f\_name) as f\_obj:

numbers = json.load(f\_obj)

except FileNotFoundError:

msg = "Can’t find {0}.".format(f\_name) print(msg)

else:

print(numbers)

Practice with exceptions

*Take a program you've already written that prompts for user input, and add some error-handling code to the program.*

*More cheat sheets available at*

**

**

**

When you write a function or a class, you can also write tests for that code. Testing proves that your code works as it's supposed to in the situations it's designed to handle, and also when people use your programs in unexpected ways. Writing tests gives you confidence that your code will work correctly as more people begin to use your programs. You can also add new features to your programs and know that you haven't broken existing behavior.

A unit test verifies that one specific aspect of your code works as it's supposed to. A test case is a collection of unit tests which verify your code's behavior in a wide variety of situations.



*Python's unittest module provides tools for testing your code. To try it out, we’ll create a function that returns a full name. We’ll use the function in a regular program, and then build a test case for the function.*

A function to test

*Save this as full\_names.py*

def get\_full\_name(first, last):

"""Return a full name."""

full\_name = "{0} {1}".format(first, last) return full\_name.title()

Using the function

*Save this as names.py*

from full\_names import get\_full\_name

janis = get\_full\_name('janis', 'joplin') print(janis)

bob = get\_full\_name('bob', 'dylan')

print(bob)

Building a testcase with one unit test

*To build a test case, make a class that inherits from* unittest.TestCase *and write methods that begin with* test\_*. Save this as test\_full\_names.py*

import unittest

from full\_names import get\_full\_name

class NamesTestCase(unittest.TestCase): """Tests for names.py."""

def test\_first\_last(self):

"""Test names like Janis Joplin.""" full\_name = get\_full\_name('janis', 'joplin')

self.assertEqual(full\_name,

'Janis Joplin')

unittest.main()

Running the test

*Python reports on each unit test in the test case. The dot reports a single passing test. Python informs us that it ran 1 test in less than 0.001 seconds, and the OK lets us know that all unit tests in the test case passed.*

.

--------------------------------------- Ran 1 test in 0.000s

OK



*Failing tests are important; they tell you that a change in the code has affected existing behavior. When a test fails, you need to modify the code so the existing behavior still works.*

Modifying the function

*We’ll modify* get\_full\_name() *so it handles middle names, but we’ll do it in a way that breaks existing behavior.*

def get\_full\_name(first, middle, last): """Return a full name."""

full\_name = "{0} {1} {2}".format(first, middle, last) return full\_name.title()

Using the function

from full\_names import get\_full\_name

john = get\_full\_name('john', 'lee', 'hooker') print(john)

david = get\_full\_name('david', 'lee', 'roth') print(david)

Running the test

*When you change your code, it’s important to run your existing tests. This will tell you whether the changes you made affected existing behavior.*

E

================================================ ERROR: test\_first\_last (\_\_main\_\_.NamesTestCase) Test names like Janis Joplin.

------------------------------------------------ Traceback (most recent call last):

File "test\_full\_names.py", line 10, in test\_first\_last

'joplin')

TypeError: get\_full\_name() missing 1 required positional argument: 'last'

------------------------------------------------ Ran 1 test in 0.001s

FAILED (errors=1)

Fixing the code

*When a test fails, the code needs to be modified until the test passes again. (Don’t make the mistake of rewriting your tests to fit your new code.) Here we can make the middle name optional.*

def get\_full\_name(first, last, middle=''): """Return a full name."""

if middle:

full\_name = "{0} {1} {2}".format(first, middle, last) else:

full\_name = "{0} {1}".format(first, last) return full\_name.title()

Running the test

*Now the test should pass again, which means our original functionality is still intact.*

.

--------------------------------------- Ran 1 test in 0.000s

OK

*Covers Python 3 and Python 2 *

**

**

**

*You can add as many unit tests to a test case as you need. To write a new test, add a new method to your test case class.*

Testing middle names

*We’ve shown that* get\_full\_name() *works for first and last names. Let’s test that it works for middle names as well.*

import unittest

from full\_names import get\_full\_name

class NamesTestCase(unittest.TestCase): """Tests for names.py."""

def test\_first\_last(self):

"""Test names like Janis Joplin.""" full\_name = get\_full\_name('janis', 'joplin')

self.assertEqual(full\_name,

'Janis Joplin')

def test\_middle(self):

"""Test names like David Lee Roth.""" full\_name = get\_full\_name('david', 'roth', 'lee')

self.assertEqual(full\_name,

'David Lee Roth')

unittest.main()

Running the tests

*The two dots represent two passing tests.*

..

--------------------------------------- Ran 2 tests in 0.000s

OK



*Python provides a number of assert methods you can use to test your code.*

Verify that a==b, or a != b

assertEqual(a, b)

assertNotEqual(a, b)

Verify that x is True, or x is False

assertTrue(x)

assertFalse(x)

Verify an item is in a list, or not in a list

assertIn(item, list)

assertNotIn(item, list)

*Testing a class is similar to testing a function, since you’ll mostly be testing your methods.*

A class to test

*Save as accountant.py*

class Accountant():

"""Manage a bank account."""

def \_\_init\_\_(self, balance=0):

self.balance = balance

def deposit(self, amount):

self.balance += amount

def withdraw(self, amount):

self.balance -= amount

Building a testcase

*For the first test, we’ll make sure we can start out with different initial balances. Save this as test\_accountant.py.*

import unittest

from accountant import Accountant

class TestAccountant(unittest.TestCase): """Tests for the class Accountant."""

def test\_initial\_balance(self):

# Default balance should be 0. acc = Accountant()

self.assertEqual(acc.balance, 0)

# Test non-default balance.

acc = Accountant(100)

self.assertEqual(acc.balance, 100)

unittest.main()

Running the test

.

--------------------------------------- Ran 1 test in 0.000s

OK



*In general you shouldn’t modify a test once it’s written. When a test fails it usually means new code you’ve written has broken existing functionality, and you need to modify the new code until all existing tests pass.*

*If your original requirements have changed, it may be appropriate to modify some tests. This usually happens in the early stages of a project when desired behavior is still being sorted out.*

*When testing a class, you usually have to make an instance of the class. The* setUp() *method is run before every test. Any instances you make in* setUp() *are available in every test you write.*

Using setUp() to support multiple tests

*The instance* self.acc *can be used in each new test.*

import unittest

from accountant import Accountant

class TestAccountant(unittest.TestCase): """Tests for the class Accountant."""

def setUp(self):

self.acc = Accountant()

def test\_initial\_balance(self):

# Default balance should be 0. self.assertEqual(self.acc.balance, 0)

# Test non-default balance.

acc = Accountant(100)

self.assertEqual(acc.balance, 100)

def test\_deposit(self):

# Test single deposit.

self.acc.deposit(100)

self.assertEqual(self.acc.balance, 100)

# Test multiple deposits.

self.acc.deposit(100)

self.acc.deposit(100)

self.assertEqual(self.acc.balance, 300)

def test\_withdrawal(self):

# Test single withdrawal.

self.acc.deposit(1000)

self.acc.withdraw(100)

self.assertEqual(self.acc.balance, 900)

unittest.main()

Running the tests

...

--------------------------------------- Ran 3 tests in 0.001s

OK

*More cheat sheets available at*

**

**

**

**

Pygame is a framework for making games using Python. Making games is fun, and it’s a great way to expand your programming skills and knowledge. Pygame takes care of many of the lower-level tasks in building games, which lets you focus on the aspects of your game that make it interesting.



*Pygame runs on all systems, but setup is slightly different on each OS. The instructions here assume you’re using Python 3, and provide a minimal installation of Pygame. If these instructions don’t work for your system, see the more detailed notes at http://ehmatthes.github.io/pcc/.*

Pygame on Linux

$ sudo apt-get install python3-dev mercurial libsdl-image1.2-dev libsdl2-dev

libsdl-ttf2.0-dev

$ pip install --user

hg+http://bitbucket.org/pygame/pygame

Pygame on OS X

*This assumes you’ve used Homebrew to install Python 3.*

$ brew install hg sdl sdl\_image sdl\_ttf $ pip install --user

hg+http://bitbucket.org/pygame/pygame

Pygame on Windows

*Find an installer at*

*https://bitbucket.org/pygame/pygame/downloads/ or http://www.lfd.uci.edu/~gohlke/pythonlibs/#pygame that matches your version of Python. Run the installer file if it’s a .exe or .msi file. If it’s a .whl file, use pip to install Pygame:*

> python –m pip install --user

pygame-1.9.2a0-cp35-none-win32.whl

Testing your installation

*To test your installation, open a terminal session and try to import Pygame. If you don’t get any error messages, your installation was successful.*

$ python

>>> import pygame

>>>

*The following code sets up an empty game window, and starts an event loop and a loop that continually refreshes the screen.*

An empty game window

import sys

import pygame as pg

def run\_game():

# Initialize and set up screen.

pg.init()

screen = pg.display.set\_mode((1200, 800)) pg.display.set\_caption("Alien Invasion")

# Start main loop.

while True:

# Start event loop.

for event in pg.event.get():

if event.type == pg.QUIT: sys.exit()

# Refresh screen.

pg.display.flip()

run\_game()

Setting a custom window size

*The display.set\_mode() function accepts a tuple that defines the screen size.*

screen\_dim = (1200, 800)

screen = pg.display.set\_mode(screen\_dim)

Setting a custom background color

*Colors are defined as a tuple of red, green, and blue values. Each value ranges from 0-255.*

bg\_color = (230, 230, 230)

screen.fill(bg\_color)



*Many objects in a game can be treated as simple rectangles, rather than their actual shape. This simplifies code without noticeably affecting game play. Pygame has a rect object that makes it easy to work with game objects.*

Getting the screen rect object

*We already have a screen object; we can easily access the rect object associated with the screen.*

screen\_rect = screen.get\_rect()

Finding the center of the screen

*Rect objects have a center attribute which stores the center point.* screen\_center = screen\_rect.center

Useful rect attributes

*Once you have a rect object, there are a number of attributes that are useful when positioning objects and detecting relative positions of objects. (You can find more attributes in the Pygame documentation.)*

# Individual x and y values:

screen\_rect.left, screen\_rect.right

screen\_rect.top, screen\_rect.bottom

screen\_rect.centerx, screen\_rect.centery screen\_rect.width, screen\_rect.height

# Tuples

screen\_rect.center

screen\_rect.size

Creating a rect object

*You can create a rect object from scratch. For example a small rect object that’s filled in can represent a bullet in a game. The Rect() class takes the coordinates of the upper left corner, and the width and height of the rect. The draw.rect() function takes a screen object, a color, and a rect. This function fills the given rect with the given color.*

bullet\_rect = pg.Rect(100, 100, 3, 15) color = (100, 100, 100)

pg.draw.rect(screen, color, bullet\_rect)



*Many objects in a game are images that are moved around the screen. It’s easiest to use bitmap (.bmp) image files, but you can also configure your system to work with jpg, png, and gif files as well.*

Loading an image

ship = pg.image.load('images/ship.bmp') Getting the rect object from an image

ship\_rect = ship.get\_rect()

Positioning an image

*With rects, it’s easy to position an image wherever you want on the screen, or in relation to another object. The following code positions a ship object at the bottom center of the screen.*

ship\_rect.midbottom = screen\_rect.midbottom*Covers Python 3 and Python 2 *

**

**

**

Drawing an image to the screen

*Once an image is loaded and positioned, you can draw it to the screen with the blit() method. The blit() method acts on the screen object, and takes the image object and image rect as arguments.*

# Draw ship to screen.

screen.blit(ship, ship\_rect)

The blitme() method

*Game objects such as ships are often written as classes. Then a blitme() method is usually defined, which draws the object to the screen.*

def blitme(self):

"""Draw ship at current location.""" self.screen.blit(self.image, self.rect)



*Pygame watches for events such as key presses and mouse actions. You can detect any event you care about in the event loop, and respond with any action that’s appropriate for your game.*

Responding to key presses

*Pygame’s main event loop registers a KEYDOWN event any time a key is pressed. When this happens, you can check for specific keys.*

for event in pg.event.get():

if event.type == pg.KEYDOWN:

if event.key == pg.K\_RIGHT:

ship\_rect.x += 1

elif event.key == pg.K\_LEFT:

ship\_rect.x -= 1

elif event.key == pg.K\_SPACE: ship.fire\_bullet()

elif event.key == pg.K\_q:

sys.exit()

Responding to released keys

*When the user releases a key, a KEYUP event is triggered.*

if event.type == pg.KEYUP:

if event.key == pg.K\_RIGHT:

ship.moving\_right = False



*The Pygame documentation is really helpful when building your own games. The home page for the Pygame project is at http://pygame.org/, and the home page for the documentation is at http://pygame.org/docs/.*

*The most useful part of the documentation are the pages about specific parts of Pygame, such as the Rect() class and the sprite module. You can find a list of these elements at the top of the help pages.*

*Pygame’s event loop registers an event any time the mouse moves, or a mouse button is pressed or released.*

Responding to the mouse button

for event in pg.event.get():

if event.type == pg.MOUSEBUTTONDOWN: ship.fire\_bullet()

Finding the mouse position

*The mouse position is returned as a tuple.*

mouse\_pos = pg.mouse.get\_pos()

Clicking a button

*You might want to know if the cursor is over an object such as a button. The rect.collidepoint() method returns true when a point is inside a rect object.*

if button\_rect.collidepoint(mouse\_pos): start\_game()

Hiding the mouse

pg.mouse.set\_visible(False)



*Pygame has a Group class which makes working with a group of similar objects easier. A group is like a list, with some extra functionality that’s helpful when building games.*

Making and filling a group

*An object that will be placed in a group must inherit from Sprite.* from pygame.sprite import Sprite, Group

def Bullet(Sprite):

...

def draw\_bullet(self):

...

def update(self):

...

bullets = Group()

new\_bullet = Bullet()

bullets.add(new\_bullet)

Looping through the items in a group

*The sprites() method returns all the members of a group.*

for bullet in bullets.sprites():

bullet.draw\_bullet()

Calling update() on a group

*Calling update() on a group automatically calls update() on each member of the group.*

bullets.update()

Removing an item from a group

*It’s important to delete elements that will never appear again in the game, so you don’t waste memory and resources.*

bullets.remove(bullet)



*You can detect when a single object collides with any member of a group. You can also detect when any member of one group collides with a member of another group.*

Collisions between a single object and a group *The spritecollideany() function takes an object and a group, and returns True if the object overlaps with any member of the group.*

if pg.sprite.spritecollideany(ship, aliens): ships\_left -= 1

Collisions between two groups

*The sprite.groupcollide() function takes two groups, and two booleans. The function returns a dictionary containing information about the members that have collided. The booleans tell Pygame whether to delete the members of either group that have collided.*

collisions = pg.sprite.groupcollide(

bullets, aliens, True, True)

score += len(collisions) \* alien\_point\_value



*You can use text for a variety of purposes in a game. For example you can share information with players, and you can display a score.*

Displaying a message

*The following code defines a message, then a color for the text and the background color for the message. A font is defined using the default system font, with a font size of 48. The font.render() function is used to create an image of the message, and we get the rect object associated with the image. We then center the image on the screen and display it.*

msg = "Play again?"

msg\_color = (100, 100, 100)

bg\_color = (230, 230, 230)

f = pg.font.SysFont(None, 48)

msg\_image = f.render(msg, True, msg\_color, bg\_color)

msg\_image\_rect = msg\_image.get\_rect() msg\_image\_rect.center = screen\_rect.center screen.blit(msg\_image, msg\_image\_rect)

*More cheat sheets available at*

**

**

**

**

Data visualization involves exploring data through visual representations. The matplotlib package helps you make visually appealing representations of the data you’re working with. matplotlib is extremely flexible; these examples will help you get started with a few simple visualizations.



*matplotlib runs on all systems, but setup is slightly different depending on your OS. If the minimal instructions here don’t work for you, see the more detailed instructions at http://ehmatthes.github.io/pcc/. You should also consider installing the Anaconda distrubution of Python from https://continuum.io/downloads/, which includes matplotlib.*

matplotlib on Linux

$ sudo apt-get install python3-matplotlib

matplotlib on OS X

*Start a terminal session and enter* import matplotlib *to see if it’s already installed on your system. If not, try this command:*

$ pip install --user matplotlib

matplotlib on Windows

*You first need to install Visual Studio, which you can do from https://dev.windows.com/. The Community edition is free. Then go to https://pypi.python.org/pypi/matplotlib/ or*

*http://www.lfd.uic.edu/~gohlke/pythonlibs/#matplotlib and download an appropriate installer file.*

**

Making a line graph

import matplotlib.pyplot as plt

x\_values = [0, 1, 2, 3, 4, 5]

squares = [0, 1, 4, 9, 16, 25]

plt.plot(x\_values, squares)

plt.show()

Making a scatter plot

*The scatter() function takes a list of x values and a list of y values, and a variety of optional arguments. The* s=10 *argument controls the size of each point.*

import matplotlib.pyplot as plt

x\_values = list(range(1000))

squares = [x\*\*2 for x in x\_values]

plt.scatter(x\_values, squares, s=10)

plt.show()



*Plots can be customized in a wide variety of ways. Just about any element of a plot can be customized.*

Adding titles and labels, and scaling axes import matplotlib.pyplot as plt

x\_values = list(range(1000))

squares = [x\*\*2 for x in x\_values]

plt.scatter(x\_values, squares, s=10)

plt.title("Square Numbers", fontsize=24) plt.xlabel("Value", fontsize=18)

plt.ylabel("Square of Value", fontsize=18) plt.tick\_params(axis='both', which='major', labelsize=14)

plt.axis([0, 1100, 0, 1100000])

plt.show()

Using a colormap

*A colormap varies the point colors from one shade to another, based on a certain value for each point. The value used to determine the color of each point is passed to the* c *argument, and the* cmap *argument specifies which colormap to use. The* edgecolor='none' *argument removes the black outline from each point.*

plt.scatter(x\_values, squares, c=squares, cmap=plt.cm.Blues, edgecolor='none', s=10)



Emphasizing points

*You can plot as much data as you want on one plot. Here we re plot the first and last points larger to emphasize them.*

import matplotlib.pyplot as plt

x\_values = list(range(1000))

squares = [x\*\*2 for x in x\_values]

plt.scatter(x\_values, squares, c=squares, cmap=plt.cm.Blues, edgecolor='none', s=10)

plt.scatter(x\_values[0], squares[0], c='green', edgecolor='none', s=100)

plt.scatter(x\_values[-1], squares[-1], c='red', edgecolor='none', s=100)

plt.title("Square Numbers", fontsize=24) *--snip--*

Removing axes

*You can customize or remove axes entirely. Here’s how to access each axis, and hide it.*

plt.axes().get\_xaxis().set\_visible(False) plt.axes().get\_yaxis().set\_visible(False)

Setting a custom figure size

*You can make your plot as big or small as you want. Before plotting your data, add the following code. The* dpi *argument is optional; if you don’t know your system’s resolution you can omit the argument and adjust the* figsize *argument accordingly.*

plt.figure(dpi=128, figsize=(10, 6))

Saving a plot

*The matplotlib viewer has an interactive save button, but you can also save your visualizations programmatically. To do so, replace* plt.show() *with* plt.savefig()*. The* bbox\_inches='tight' *argument trims extra whitespace from the plot.*

plt.savefig('squares.png', bbox\_inches='tight')



*The matplotlib gallery and documentation are at http://matplotlib.org/. Be sure to visit the examples, gallery, and pyplot links.*

*Covers Python 3 and Python 2 *

**

**

*You can make as many plots as you want on one figure. When you make multiple plots, you can emphasize relationships in the data. For example you can fill the space between two sets of data.*

Plotting two sets of data

*Here we use* plt.scatter() *twice to plot square numbers and cubes on the same figure.*

import matplotlib.pyplot as plt

x\_values = list(range(11))

squares = [x\*\*2 for x in x\_values]

cubes = [x\*\*3 for x in x\_values]

plt.scatter(x\_values, squares, c='blue', edgecolor='none', s=20)

plt.scatter(x\_values, cubes, c='red', edgecolor='none', s=20)

plt.axis([0, 11, 0, 1100])

plt.show()

Filling the space between data sets

*The* fill\_between() *method fills the space between two data sets. It takes a series of x-values and two series of y-values. It also takes a* facecolor *to use for the fill, and an optional* alpha *argument that controls the color’s transparency.*

plt.fill\_between(x\_values, cubes, squares, facecolor='blue', alpha=0.25)



*Many interesting data sets have a date or time as the x value. Python’s datetime module helps you work with this kind of data.*

Generating the current date

*The* datetime.now() *function returns a datetime object representing the current date and time.*

from datetime import datetime as dt

today = dt.now()

date\_string = dt.strftime(today, '%m/%d/%Y') print(date\_string)

Generating a specific date

*You can also generate a datetime object for any date and time you want. The positional order of arguments is year, month, and day. The hour, minute, second, and microsecond arguments are optional.*

from datetime import datetime as dt

new\_years = dt(2017, 1, 1)

fall\_equinox = dt(year=2016, month=9, day=22)

Datetime formatting arguments

*The* strftime() *function generates a formatted string from a datetime object, and the* strptime() *function genereates a datetime object from a string. The following codes let you work with dates exactly as you need to.*

%A Weekday name, such as Monday

%B Month name, such as January

%m Month, as a number (01 to 12)

%d Day of the month, as a number (01 to 31) %Y Four-digit year, such as 2016

%y Two-digit year, such as 16

%H Hour, in 24-hour format (00 to 23) %I Hour, in 12-hour format (01 to 12) %p AM or PM

%M Minutes (00 to 59)

%S Seconds (00 to 61)

Converting a string to a datetime object

new\_years = dt.strptime('1/1/2017', '%m/%d/%Y') Converting a datetime object to a string

ny\_string = dt.strftime(new\_years, '%B %d, %Y') print(ny\_string)

Plotting high temperatures

*The following code creates a list of dates and a corresponding list of high temperatures. It then plots the high temperatures, with the date labels displayed in a specific format.*

from datetime import datetime as dt

import matplotlib.pyplot as plt

from matplotlib import dates as mdates

dates = [

dt(2016, 6, 21), dt(2016, 6, 22), dt(2016, 6, 23), dt(2016, 6, 24), ]

highs = [57, 68, 64, 59]

fig = plt.figure(dpi=128, figsize=(10,6)) plt.plot(dates, highs, c='red')

plt.title("Daily High Temps", fontsize=24) plt.ylabel("Temp (F)", fontsize=16)

x\_axis = plt.axes().get\_xaxis()

x\_axis.set\_major\_formatter(

mdates.DateFormatter('%B %d %Y') )

fig.autofmt\_xdate()

plt.show()

*You can include as many individual graphs in one figure as you want. This is useful, for example, when comparing related datasets.*

Sharing an x-axis

*The following code plots a set of squares and a set of cubes on two separate graphs that share a common x-axis.*

*The* plt.subplots() *function returns a figure object and a tuple of axes. Each set of axes corresponds to a separate plot in the figure. The first two arguments control the number of rows and columns generated in the figure.*

import matplotlib.pyplot as plt

x\_vals = list(range(11))

squares = [x\*\*2 for x in x\_vals]

cubes = [x\*\*3 for x in x\_vals]

fig, axarr = plt.subplots(2, 1, sharex=True)

axarr[0].scatter(x\_vals, squares)

axarr[0].set\_title('Squares')

axarr[1].scatter(x\_vals, cubes, c='red') axarr[1].set\_title('Cubes')

plt.show()

Sharing a y-axis

*To share a y-axis, we use the* sharey=True *argument.* import matplotlib.pyplot as plt

x\_vals = list(range(11))

squares = [x\*\*2 for x in x\_vals]

cubes = [x\*\*3 for x in x\_vals]

fig, axarr = plt.subplots(1, 2, sharey=True)

axarr[0].scatter(x\_vals, squares)

axarr[0].set\_title('Squares')

axarr[1].scatter(x\_vals, cubes, c='red') axarr[1].set\_title('Cubes')

plt.show()



*More cheat sheets available at*

**

**

**

**

Data visualization involves exploring data through visual representations. Pygal helps you make visually appealing representations of the data you’re working with. Pygal is particularly well suited for visualizations that will be presented online, because it supports interactive elements.



*Pygal can be installed using pip.*

Pygal on Linux and OS X

$ pip install --user pygal

Pygal on Windows

> python –m pip install --user pygal

*To make a plot with Pygal, you specify the kind of plot and then add the data.*

Making a line graph

*To view the output, open the file squares.svg in a browser.* import pygal

x\_values = [0, 1, 2, 3, 4, 5]

squares = [0, 1, 4, 9, 16, 25]

chart = pygal.Line()

chart.force\_uri\_protocol = 'http'

chart.add('x^2', squares)

chart.render\_to\_file('squares.svg')

Adding labels and a title

*--snip--*

chart = pygal.Line()

chart.force\_uri\_protocol = 'http'

chart.title = "Squares"

chart.x\_labels = x\_values

chart.x\_title = "Value"

chart.y\_title = "Square of Value"

chart.add('x^2', squares)

chart.render\_to\_file('squares.svg')

Making a scatter plot

*The data for a scatter plot needs to be a list containing tuples of the form (x, y). The* stroke=False *argument tells Pygal to make an XY chart with no line connecting the points.*

import pygal

squares = [

(0, 0), (1, 1), (2, 4), (3, 9),

(4, 16), (5, 25),

]

chart = pygal.XY(stroke=False)

chart.force\_uri\_protocol = 'http'

chart.add('x^2', squares)

chart.render\_to\_file('squares.svg')

Using a list comprehension for a scatter plot *A list comprehension can be used to effficiently make a dataset for a scatter plot.*

squares = [(x, x\*\*2) for x in range(1000)]

Making a bar graph

*A bar graph requires a list of values for the bar sizes. To label the bars, pass a list of the same length to* x\_labels*.*

import pygal

outcomes = [1, 2, 3, 4, 5, 6]

frequencies = [18, 16, 18, 17, 18, 13]

chart = pygal.Bar()

chart.force\_uri\_protocol = 'http'

chart.x\_labels = outcomes

chart.add('D6', frequencies)

chart.render\_to\_file('rolling\_dice.svg')

Making a bar graph from a dictionary

*Since each bar needs a label and a value, a dictionary is a great way to store the data for a bar graph. The keys are used as the labels along the x-axis, and the values are used to determine the height of each bar.*

import pygal

results = {

1:18, 2:16, 3:18,

4:17, 5:18, 6:13,

}

chart = pygal.Bar()

chart.force\_uri\_protocol = 'http'

chart.x\_labels = results.keys()

chart.add('D6', results.values())

chart.render\_to\_file('rolling\_dice.svg')

*You can add as much data as you want when making a visualization.*

Plotting squares and cubes

import pygal

x\_values = list(range(11))

squares = [x\*\*2 for x in x\_values]

cubes = [x\*\*3 for x in x\_values]

chart = pygal.Line()

chart.force\_uri\_protocol = 'http'

chart.title = "Squares and Cubes"

chart.x\_labels = x\_values

chart.add('Squares', squares)

chart.add('Cubes', cubes)

chart.render\_to\_file('squares\_cubes.svg')

Filling the area under a data series

*Pygal allows you to fill the area under or over each series of data. The default is to fill from the x-axis up, but you can fill from any horizontal line using the* zero *argument.*

chart = pygal.Line(fill=True, zero=0) 



*The documentation for Pygal is available at*

*http://www.pygal.org/.*

**

*If you’re viewing svg output in a browser, Pygal needs to render the output file in a specific way. The*

force\_uri\_protocol *attribute for chart objects needs to be set to* 'http'.

*Covers Python 3 and Python 2 *

**

**

*Pygal lets you customize many elements of a plot. There are some excellent default themes, and many options for styling individual plot elements.*

Using built-in styles

*To use built-in styles, import the style and make an instance of the style class. Then pass the style object with the* style *argument when you make the chart object.*

import pygal

from pygal.style import LightGreenStyle

x\_values = list(range(11))

squares = [x\*\*2 for x in x\_values]

cubes = [x\*\*3 for x in x\_values]

chart\_style = LightGreenStyle()

chart = pygal.Line(style=chart\_style) chart.force\_uri\_protocol = 'http'

chart.title = "Squares and Cubes"

chart.x\_labels = x\_values

chart.add('Squares', squares)

chart.add('Cubes', cubes)

chart.render\_to\_file('squares\_cubes.svg')

Parametric built-in styles

*Some built-in styles accept a custom color, then generate a theme based on that color.*

from pygal.style import LightenStyle

*--snip--*

chart\_style = LightenStyle('#336688') chart = pygal.Line(style=chart\_style) *--snip--*

Customizing individual style properties

*Style objects have a number of properties you can set individually.*

chart\_style = LightenStyle('#336688') chart\_style.plot\_background = '#CCCCCC' chart\_style.major\_label\_font\_size = 20 chart\_style.label\_font\_size = 16

*--snip--*

Custom style class

Configuration settings

*Some settings are controlled by a* Config *object.*

my\_config = pygal.Config()

my\_config.show\_y\_guides = False

my\_config.width = 1000

my\_config.dots\_size = 5

chart = pygal.Line(config=my\_config) *--snip--*

Styling series

*You can give each series on a chart different style settings.*

chart.add('Squares', squares, dots\_size=2) chart.add('Cubes', cubes, dots\_size=3)

Styling individual data points

*You can style individual data points as well. To do so, write a dictionary for each data point you want to customize. A* 'value' *key is required, and other properies are optional.*

import pygal

repos = [

{

'value': 20506,

'color': '#3333CC',

'xlink': 'http://djangoproject.com/', },

20054,

12607,

11827,

]

chart = pygal.Bar()

chart.force\_uri\_protocol = 'http'

chart.x\_labels = [

'django', 'requests', 'scikit-learn', 'tornado',

]

chart.y\_title = 'Stars'

chart.add('Python Repos', repos)

chart.render\_to\_file('python\_repos.svg')

*Pygal can generate world maps, and you can add any data you want to these maps. Data is indicated by coloring, by labels, and by tooltips that show data when users hover over each country on the map.*

Installing the world map module

*The world map module is not included by default in Pygal 2.0. It can be installed with pip:*

$ pip install --user pygal\_maps\_world

Making a world map

*The following code makes a simple world map showing the countries of North America.*

from pygal.maps.world import World

wm = World()

wm.force\_uri\_protocol = 'http'

wm.title = 'North America'

wm.add('North America', ['ca', 'mx', 'us'])

wm.render\_to\_file('north\_america.svg')

Showing all the country codes

*In order to make maps, you need to know Pygal’s country codes. The following example will print an alphabetical list of each country and its code.*

from pygal.maps.world import COUNTRIES

for code in sorted(COUNTRIES.keys()): print(code, COUNTRIES[code])

Plotting numerical data on a world map

*To plot numerical data on a map, pass a dictionary to* add() *instead of a list.*

from pygal.maps.world import World

populations = {

'ca': 34126000,

'us': 309349000,

'mx': 113423000,

}

wm = World()

wm.force\_uri\_protocol = 'http'

*You can start with a bare style class, and then set only the properties you care about.*

chart\_style = Style()

chart\_style.colors = [

'#CCCCCC', '#AAAAAA', '#888888'] chart\_style.plot\_background = '#EEEEEE'

chart = pygal.Line(style=chart\_style) *--snip--*

**

wm.title = 'Population of North America'

wm.add('North America', populations)

wm.render\_to\_file('na\_populations.svg')

*More cheat sheets available at*

**

**

**

Django is a web framework which helps you build interactive websites using Python. With Django you define the kind of data your site needs to work with, and you define the ways your users can work with that data.



*It’s usualy best to install Django to a virtual environment, where your project can be isolated from your other Python projects. Most commands assume you’re working in an active virtual environment.*

Create a virtual environment

$ python –m venv ll\_env

Activate the environment (Linux and OS X) $ source ll\_env/bin/activate

Activate the environment (Windows)

> ll\_env\Scripts\activate

Install Django to the active environment

(ll\_env)$ pip install Django



*To start a project we’ll create a new project, create a database, and start a development server.*

Create a new project

$ django-admin.py startproject learning\_log . Create a database

$ python manage.py migrate

View the project

*After issuing this command, you can view the project at http://localhost:8000/.*

$ python manage.py runserver

Create a new app

*A Django project is made up of one or more apps.* $ python manage.py startapp learning\_logs

*The data in a Django project is structured as a set of models.*

Defining a model

*To define the models for your app, modify the file models.py that was created in your app’s folder. The* \_\_str\_\_() *method tells Django how to represent data objects based on this model.*

from django.db import models

class Topic(models.Model):

"""A topic the user is learning about.""" text = models.CharField(max\_length=200) date\_added = models.DateTimeField( auto\_now\_add=True)

def \_\_str\_\_(self):

return self.text

Activating a model

*To use a model the app must be added to the tuple* INSTALLED\_APPS*, which is stored in the project’s settings.py file.*

INSTALLED\_APPS = (

*--snip--*

'django.contrib.staticfiles',

# My apps

'learning\_logs',

)

Migrating the database

*The database needs to be modified to store the kind of data that the model represents.*

$ python manage.py makemigrations learning\_logs $ python manage.py migrate

Creating a superuser

*A superuser is a user account that has access to all aspects of the project.*

$ python manage.py createsuperuser

Registering a model

*You can register your models with Django’s admin site, which makes it easier to work with the data in your project. To do this, modify the app’s admin.py file. View the admin site at http://localhost:8000/admin/.*

from django.contrib import admin

from learning\_logs.models import Topic admin.site.register(Topic)

*Users interact with a project through web pages, and a project’s home page can start out as a simple page with no data. A page usually needs a URL, a view, and a template.*

Mapping a project’s URLs

*The project’s main urls.py file tells Django where to find the urls.py files associated with each app in the project.*

from django.conf.urls import include, url from django.contrib import admin

urlpatterns = [

url(r'^admin/', include(admin.site.urls)), url(r'', include('learning\_logs.urls', namespace='learning\_logs')), ]

Mapping an app’s URLs

*An app’s urls.py file tells Django which view to use for each URL in the app. You’ll need to make this file yourself, and save it in the app’s folder.*

from django.conf.urls import url

from . import views

urlpatterns = [

url(r'^$', views.index, name='index'), ]

Writing a simple view

*A view takes information from a request and sends data to the browser, often through a template. View functions are stored in an app’s views.py file. This simple view function doesn’t pull in any data, but it uses the template index.html to render the home page.*

from django.shortcuts import render

def index(request):

"""The home page for Learning Log.""" return render(request,

'learning\_logs/index.html')



*The documentation for Django is available at http://docs.djangoproject.com/. The Django documentation is thorough and user-friendly, so check it out!*

**

*Covers Python 3 and Python 2*

**

**

Writing a simple template

*A template sets up the structure for a page. It’s a mix of html and template code, which is like Python but not as powerful. Make a folder called templates inside the project folder. Inside the templates folder make another folder with the same name as the app. This is where the template files should be saved.*

<p>Learning Log</p>

<p>Learning Log helps you keep track of your learning, for any topic you're learning about.</p>



*Many elements of a web page are repeated on every page in the site, or every page in a section of the site. By writing one parent template for the site, and one for each section, you can easily modify the look and feel of your entire site.*

The parent template

*The parent template defines the elements common to a set of pages, and defines blocks that will be filled by individual pages.*

<p>

<a href="{% url 'learning\_logs:index' %}"> Learning Log

</a>

</p>

{% block content %}{% endblock content %}

The child template

*The child template uses the* {% extends %} *template tag to pull in the structure of the parent template. It then defines the content for any blocks defined in the parent template.*

{% extends 'learning\_logs/base.html' %}

{% block content %}

<p>

Learning Log helps you keep track of your learning, for any topic you're learning about.

</p>

{% endblock content %}



*Python code is usually indented by four spaces. In templates you’ll often see two spaces used for indentation, because elements tend to be nested more deeply in templates.*

*A new model can use an existing model. The ForeignKey attribute establishes a connection between instances of the two related models. Make sure to migrate the database after adding a new model to your app.*

Defining a model with a foreign key

class Entry(models.Model):

"""Learning log entries for a topic.""" topic = models.ForeignKey(Topic)

text = models.TextField()

date\_added = models.DateTimeField( auto\_now\_add=True)

def \_\_str\_\_(self):

return self.text[:50] + "..."



*Most pages in a project need to present data that’s specific to the current user.*

URL parameters

*A URL often needs to accept a parameter telling it which data to access from the database. The second URL pattern shown here looks for the ID of a specific topic and stores it in the parameter* topic\_id*.*

urlpatterns = [

url(r'^$', views.index, name='index'), url(r'^topics/(?P<topic\_id>\d+)/$', views.topic, name='topic'), ]

Using data in a view

*The view uses a parameter from the URL to pull the correct data from the database. In this example the view is sending a* context *dictionary to the template, containing data that should be displayed on the page.*

def topic(request, topic\_id):

"""Show a topic and all its entries.""" topic = Topics.objects.get(id=topic\_id) entries = topic.entry\_set.order\_by( '-date\_added')

context = {

'topic': topic,

'entries': entries,

}

return render(request,

'learning\_logs/topic.html', context)



*If you make a change to your project and the change doesn’t seem to have any effect, try restarting the server:* $ python manage.py runserver

Using data in a template

*The data in the view function’s* context *dictionary is available within the template. This data is accessed using template variables, which are indicated by doubled curly braces. The vertical line after a template variable indicates a filter. In this case a filter called* date *formats date objects, and the filter* linebreaks *renders paragraphs properly on a web page.*

{% extends 'learning\_logs/base.html' %} {% block content %}

<p>Topic: {{ topic }}</p>

<p>Entries:</p>

<ul>

{% for entry in entries %}

<li>

<p>

{{ entry.date\_added|date:'M d, Y H:i' }} </p>

<p>

{{ entry.text|linebreaks }}

</p>

</li>

{% empty %}

<li>There are no entries yet.</li> {% endfor %}

</ul>

{% endblock content %}



*You can explore the data in your project from the command line. This is helpful for developing queries and testing code snippets.*

Start a shell session

$ python manage.py shell

Access data from the project

>>> from learning\_logs.models import Topic >>> Topic.objects.all()

[<Topic: Chess>, <Topic: Rock Climbing>] >>> topic = Topic.objects.get(id=1)

>>> topic.text

'Chess'

*More cheat sheets available at*

**

**

**

**

Most web applications need to let users create accounts. This lets users create and work with their own data. Some of this data may be private, and some may be public. Django’s forms allow users to enter and modify their data.



*User accounts are handled by a dedicated app called* users*. Users need to be able to register, log in, and log out. Django automates much of this work for you.*

Making a users app

*After making the app, be sure to add* 'users' *to* INSTALLED\_APPS *in the project’s settings.py file.*

$ python manage.py startapp users

Including URLS for the users app

*Add a line to the project’s urls.py file so the* users *app’s URLs are included in the project.*

urlpatterns = [

url(r'^admin/', include(admin.site.urls)), url(r'^users/', include('users.urls', namespace='users')),

url(r'', include('learning\_logs.urls', namespace='learning\_logs')), ]



*There are a number of ways to create forms and work with them. You can use Django’s defaults, or completely customize your forms. For a simple way to let users enter data based on your models, use a* ModelForm*. This creates a form that allows users to enter data that will populate the fields on a model.*

*The register view on the back of this sheet shows a simple approach to form processing. If the view doesn’t receive data from a form, it responds with a blank form. If it receives* POST *data from a form, it validates the data and then saves it to the database.*

Defining the URLs

*Users will need to be able to log in, log out, and register. Make a new urls.py file in the* users *app folder. The login view is a default view provided by Django.*

from django.conf.urls import url

from django.contrib.auth.views import login from . import views

urlpatterns = [

url(r'^login/$', login,

{'template\_name': 'users/login.html'}, name='login'),

url(r'^logout/$', views.logout\_view, name='logout'),

url(r'^register/$', views.register, name='register'),

]

The login template

*The login view is provided by default, but you need to provide your own login template. The template shown here displays a simple login form, and provides basic error messages. Make a templates folder in the users folder, and then make a users folder in the templates folder. Save this file as login.html.*

*The tag* {% csrf\_token %} *helps prevent a common type of attack with forms. The* {{ form.as\_p }} *element displays the default login form in paragraph format. The* <input> *element named* next *redirects the user to the home page after a successful login.*

{% extends "learning\_logs/base.html" %}

{% block content %}

{% if form.errors %}

<p>

Your username and password didn't match. Please try again.

</p>

{% endif %}

<form method="post"

action="{% url 'users:login' %}"> {% csrf token %}

{{ form.as\_p }}

<button name="submit">log in</button>

<input type="hidden" name="next"

value="{% url 'learning\_logs:index' %}"/> </form>

{% endblock content %}

Showing the current login status

*You can modify the base.html template to show whether the user is currently logged in, and to provide a link to the login and logout pages. Django makes a* user *object available to every template, and this template takes advantage of this object.*

*The* user.is\_authenticated *tag allows you to serve specific content to users depending on whether they have logged in or not. The* {{ user.username }} *property allows you to greet users who have logged in. Users who haven’t logged in see links to register or log in.*

<p>

<a href="{% url 'learning\_logs:index' %}"> Learning Log

</a>

{% if user.is\_authenticated %}

Hello, {{ user.username }}.

<a href="{% url 'users:logout' %}"> log out

</a>

{% else %}

<a href="{% url 'users:register' %}"> register

</a> -

<a href="{% url 'users:login' %}"> log in

</a>

{% endif %}

</p>

{% block content %}{% endblock content %}

The logout view

*The* logout\_view() *function uses Django’s* logout() *function and then redirects the user back to the home page. Since there is no logout page, there is no logout template. Make sure to write this code in the views.py file that’s stored in the users app folder.*

from django.http import HttpResponseRedirect from django.core.urlresolvers import reverse from django.contrib.auth import logout

def logout\_view(request):

"""Log the user out."""

logout(request)

return HttpResponseRedirect(

reverse('learning\_logs:index')) *Covers Python 3 and Python 2 *

**

**

**

The register view

*The register view needs to display a blank registration form when the page is first requested, and then process completed registration forms. A successful registration logs the user in and redirects to the home page.*

from django.contrib.auth import login from django.contrib.auth import authenticate from django.contrib.auth.forms import \ UserCreationForm

def register(request):

"""Register a new user."""

if request.method != 'POST':

# Show blank registration form. form = UserCreationForm()

else:

# Process completed form.

form = UserCreationForm(

data=request.POST)

if form.is\_valid():

new\_user = form.save()

# Log in, redirect to home page. pw = request.POST['password1'] authenticated\_user = authenticate( username=new\_user.username, password=pw

)

login(request, authenticated\_user) return HttpResponseRedirect( reverse('learning\_logs:index'))

context = {'form': form}

return render(request,

'users/register.html', context)



*The django-bootstrap3 app allows you to use the Bootstrap library to make your project look visually appealing. The app provides tags that you can use in your templates to style individual elements on a page. Learn more at http://django-bootstrap3.readthedocs.io/.*

**

*Heroku lets you push your project to a live server, making it available to anyone with an internet connection. Heroku offers a free service level, which lets you learn the deployment process without any commitment. You’ll need to install a set of heroku tools, and use git to track the state of your project. See http://devcenter.heroku.com/, and click on the Python link.*

The register template

*The register template displays the registration form in paragraph formats.*

{% extends 'learning\_logs/base.html' %} {% block content %}

<form method='post'

action="{% url 'users:register' %}">

{% csrf\_token %}

{{ form.as\_p }}

<button name='submit'>register</button> <input type='hidden' name='next'

value="{% url 'learning\_logs:index' %}"/>

</form>

{% endblock content %}



*Users will have data that belongs to them. Any model that should be connected directly to a user needs a field connecting instances of the model to a specific user.*

Making a topic belong to a user

*Only the highest-level data in a hierarchy needs to be directly connected to a user. To do this import the* User *model, and add it as a foreign key on the data model.*

*After modifying the model you’ll need to migrate the database. You’ll need to choose a user ID to connect each existing instance to.*

from django.db import models

from django.contrib.auth.models import User

class Topic(models.Model):

"""A topic the user is learning about.""" text = models.CharField(max\_length=200) date\_added = models.DateTimeField( auto\_now\_add=True)

owner = models.ForeignKey(User)

def \_\_str\_\_(self):

return self.text

Querying data for the current user

*In a view, the request object has a* user *attribute. You can use this attribute to query for the user’s data. The* filter() *function then pulls the data that belongs to the current user.*

topics = Topic.objects.filter(

owner=request.user)

Restricting access to logged-in users

*Some pages are only relevant to registered users. The views for these pages can be protected by the* @login\_required *decorator. Any view with this decorator will automatically redirect non-logged in users to an appropriate page. Here’s an example views.py file.*

from django.contrib.auth.decorators import / login\_required

*--snip--*

@login\_required

def topic(request, topic\_id):

"""Show a topic and all its entries."""

Setting the redirect URL

*The* @login\_required *decorator sends unauthorized users to the login page. Add the following line to your project’s settings.py file so Django will know how to find your login page.*

LOGIN\_URL = '/users/login/'

Preventing inadvertent access

*Some pages serve data based on a parameter in the URL. You can check that the current user owns the requested data, and return a 404 error if they don’t. Here’s an example view.*

from django.http import Http404

*--snip--*

def topic(request, topic\_id):

"""Show a topic and all its entries.""" topic = Topics.objects.get(id=topic\_id) if topic.owner != request.user:

raise Http404

*--snip--*

**

*If you provide some initial data, Django generates a form with the user’s existing data. Users can then modify and save their data.*

Creating a form with initial data

*The instance parameter allows you to specify initial data for a form.* form = EntryForm(instance=entry)

Modifying data before saving

*The argument* commit=False *allows you to make changes before writing data to the database.*

new\_topic = form.save(commit=False)

new\_topic.owner = request.user

new\_topic.save()

*More cheat sheets available at*

**