|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | |  | | --- | |  | | |  | | --- | |  | |
|  |
|  | |  | | --- | | List comprehensions | | *Dictionaries store connections between pieces of*  *information. Each item in a dictionary is a key-value pair.* |
|  | |  | | --- | | squares = [x\*\*2 for x in range(1, 11)] | | |  | | --- | | A simple dictionary | |
|  |  |  |
|  | |  | | --- | | Slicing a list | | |  | | --- | | alien = {'color': 'green', 'points': 5} | |
|  | |  | | --- | | finishers = ['sam', 'bob', 'ada', 'bea'] | | |  | | --- | | Accessing a value | |
| first\_two = finishers[:2] |
|  |  |  |
| *Variables are used to store values. A string is a series of* | |  | | --- | | Copying a list | | |  | | --- | | print("The alien's color is " + alien['color']) | |
| *characters, surrounded by single or double quotes.* |  |  |
|  | |  | | --- | |  |   pyof\_ke= bke:] | |  | | --- | | Adding a new key-value pair | |
| Hello world | pyof\_ke= bke:] | |  | | --- | | alien['x\_position'] = 0 | |
| print("Hello world!") | |  | | --- | |  | | |  | | --- | | Looping through all key-value pairs | |
|  |  | |  | | --- | | fav\_numbers = {'eric': 17, 'ever': 4} | |
| Hello world with a variable | *Tuples are similar to lists, but the items in a tuple can't be* |
| msg = "Hello world!" | *modified.* | for name, number in fav\_numbers.items(): |
|  |
|  | |  | | --- | | Making a tuple | | print(name + ' loves ' + str(number)) |
| print(msg) |
|  |  |  |
| Concatenation (combining strings) | |  | | --- | | dimensions = (1920, 1080) | | |  | | --- | | Looping through all keys | |
| first\_name = 'albert' | |  | | --- | |  | | |  | | --- | | fav\_numbers = {'eric': 17, 'ever': 4} | |
| for name in fav\_numbers.keys(): |
| last\_name = 'einstein' |
| print(name + ' loves a number') |
| full\_name = first\_name + ' ' + last\_name | *If statements are used to test for particular conditions and* |
| print(full\_name) | *respond appropriately.* | |  | | --- | | Looping through all the values | |
|  | |  | | --- | | Conditional tests | | |  | | --- | | fav\_numbers = {'eric': 17, 'ever': 4} | |
|  |
|  | equals x == 42 |  |
| |  | | --- | | *A list stores a series of items in a particular order. You access items using an index, or within a loop.* | | print(str(number) + ' is a favorite') |
| not equal x != 42 |
| greater than x > 42 |
| |  | | --- | | Make a list | |  | |  | | --- | |  | |
| or equal to x >= 42 |
|  |  |
| bikes = ['trek', 'redline', 'giant'] | less than x < 42 | |  | | --- | | *Your programs can prompt the user for input. All input is stored as a string.* | |
| or equal to x <= 42 |
|  |  |  |
| |  | | --- | | Get the first item in a list | | |  | | --- | | Conditional test with lists | | |  | | --- | | Prompting for a value | |
|  |  |  |
| first\_bike = bikes[0] | 'trek' in bikes | name = input("What's your name? ") |
| |  | | --- | | Get the last item in a list | | 'surly' not in bikes |  |
| print("Hello, " + name + "!") |
|  |  |  |
| last\_bike = bikes[-1] | |  | | --- | | Assigning boolean values | | |  | | --- | | Prompting for numerical input | |
| |  | | --- | | Looping through a list | | game\_active = True | |  | | --- | | age = input("How old are you? ") | |
| can\_edit = False |
| fr bi in bikes: |  | age = int(age) |
| fr bi in bikes: | |  | | --- | | A simple if test | | pi = input("What's the value of pi? ") |
| print(bike) |
| |  | | --- | | Adding items to a list | | if age >= 18: |  |
| pi = float(pi) |
| int"Y cn ve! |
|  | int"Y cn ve! |  |
| bikes = [] | |  | | --- | | If-elif-else statements | |  |
| bikes.append('trek') |
| bikes.append('redline') | if age < 4: |  |
| bikes.append('giant') |
| ticket\_price = 0 |
| |  | | --- | | Making numerical lists | | elif age < 18: |  |
| cke\_pce= 1 |
| squares = [] | cke\_pce= 1 |  |
| else: |
| for x in range(1, 11): | ticket\_price = 15 |
| squares.append(x\*\*2) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | |  | | --- | |  | | |  | | --- | |  | |
| *A while loop repeats a block of code as long as a certain condition is true.* | *A class defines the behavior of an object and the kind of information an object can store. The information in a class* | *Your programs can read from files and write to files. Files are opened in read mode ('r') by default, but can also be* |
| A simple while loop | *is stored in attributes, and functions that belong to a class* | *opened in write mode ('w') and append mode ('a').* |
|  | *are called methods. A child class inherits the attributes and* |  |
| current\_value = 1 | *methods from its parent class.* | |  | | --- | | Reading a file and storing its lines | |
| while current\_value <= 5: | |  | | --- | | Creating a dog class | | |  | | --- | | filename = 'siddhartha.txt' | |
| print(current\_value) | with open(filename) as file\_object: |
| current\_value += 1 | class Dog(): | lines = file\_object.readlines() |
|  |
| Letting the user choose when to quit | |  | | --- | | """Represent a dog.""" | | for line in lines: |
| msg = '' | def \_\_init\_\_(self, name): | print(line) |
|  |
| while msg != 'quit': | """Initialize dog object.""" | |  | | --- | | Writing to a file | |
| self.name = name |
| msg = input("What's your message? ") |
|  | def sit(self): | |  | | --- | | filename = 'journal.txt' | |
| print(msg) |
|  |
|  |  |  |
| |  | | --- | |  | | """Simulate sitting.""" |  |
| file\_object.write("I love programming.") |
|  |
| *Functions are named blocks of code, designed to do one* | print(self.name + " is sitting.") | |  | | --- | | Appending to a file | |
| *specific job. Information passed to a function is called an argument, and information received by a function is called a parameter.* | my\_dog = Dog('Peso') | |  | | --- | | filename = 'journal.txt' | |
| print(my\_dog.name + " is a great dog!") | with open(filename, 'a') as file\_object: |
| file\_object.write("\nI love making games.") |
| |  | | --- | | A simple function | | my\_dog.sit() |  |
| def greet\_user(): | |  | | --- | | Inheritance | | |  | | --- | |  | |
| """Display a simple greeting.""" | class SARDog(Dog): | *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* |
| print("Hello!") | """Represent a search dog.""" |
| greet\_user() | |  | | --- | | def \_\_init\_\_(self, name): | |
| |  | | --- | | Passing an argument | | """Initialize the sardog.""" | *if the try block was successful goes in the else block.* |
|  |
| def greet\_user(username): | super().\_\_init\_\_(name) | |  | | --- | | Catching an exception | |
| """Display a personalized greeting.""" | def search(self): | |  | | --- | | prompt = "How many tickets do you need? " | |
| print("Hello, " + username + "!") | """Simulate searching.""" |
| num\_tickets = input(prompt) |
| greet\_user('jesse') | print(self.name + " is searching.") |
| try: |
|  |
| |  | | --- | | Default values for parameters | | my\_dog = SARDog('Willie') | num\_tickets = int(num\_tickets) |
| except ValueError: |
| def make\_pizza(topping='bacon'): | print(my\_dog.name + " is a search dog.") |  |
| print("Please try again.") |
| """Make a single-topping pizza.""" | my\_dog.sit() | else: |
| print("Have a " + topping + " pizza!") | my\_dog.search() |
| print("Your tickets are printing.") |
| make\_pizza() | |  | | --- | |  | | |  | | --- | |  | |
| make\_pizza('pepperoni') |
| *If you had infinite programming skills, what would you* |  |
| |  | | --- | | Returning a value | | *build?* | *Simple is better than complex* |
|  |  |  |
| def add\_numbers(x, y): | 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 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. | |
| """Add two numbers and return the sum.""" |
| return x + y |
| sum = add\_numbers(3, 5) |  |  |
|  | If you haven't done so already, take a few minutes and describe three projects you'd like to create. | *More cheat sheets available at* |
| print(sum) |
|  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | |  | | --- | |  | | |  | | --- | |  | |
| *You can add elements to the end of a list, or you can insert them wherever you like in a list.* | *The sort() method changes the order of a list permanently. The sorted() function returns a copy of the list, leaving the* |
|  |
|  | |  | | --- | | Adding an element to the end of the list | | *original list unchanged. You can sort the items in a list in alphabetical order, or reverse alphabetical order. You can* |
|  | |  | | --- | | users.append('amy') | | *also reverse the original order of the list. Keep in mind that* |
|  | |  | | --- | | Starting with an empty list | | *lowercase and uppercase letters may affect the sort order.* |
| |  | | --- | | Sorting a list permanently | |
|  |  |
|  | |  | | --- | | users = [] | |  |
| A list stores a series of items in a particular order. |  | |  | | --- | | users.sort() | |
| users.append('val') |
| Lists allow you to store sets of information in one place, whether you have just a few items or millions | users.append('bob') | |  | | --- | | Sorting a list permanently in reverse alphabetical order | |
| users.append('mia') |
|  |  |  |
| features readily accessible to new programmers, and | |  | | --- | | Inserting elements at a particular position | | |  | | --- | | users.sort(reverse=True) | |
| they tie together many important concepts in programming. | |  | | --- | | users.insert(0, 'joe') | | |  | | --- | | Sorting a list temporarily | |
| users.insert(3, 'bea') |
|  |  | |  | | --- | | print(sorted(users)) | |
|  | |  | | --- | |  | | print(sorted(users, reverse=True)) |
|  |  | |  | | --- | | Reversing the order of a list | |
| *You can remove elements by their position in a list, or by* |
| *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.* | *the value of the item. If you remove an item by its value, Python removes only the first item that has that value.* | |  | | --- | | users.reverse() | |
|  |  |  |
| Making a list | |  | | --- | | Deleting an element by its position | | |  | | --- | |  | |
|  |  |  |
| users = ['val', 'bob', 'mia', 'ron', 'ned'] | |  | | --- | | del users[-1] | | |  | | --- | | *Lists can contain millions of items, so Python provides an* | |
|  |  |  |
|  | |  | | --- | | Removing an item by its value | | |  | | --- | | *you set up a loop, Python pulls each item from the list one* | |
|  | |  | | --- | | users.remove('mia') | | |  | | --- | | *at a time and stores it in a temporary variable, which you provide a name for. This name should be the singular* | |
|  |
| *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* | |  | | --- | |  | | |  | | --- | | *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.* | |
| *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* |
| *index of the element in square brackets.* | *list as a stack of items, pop() takes an item off the top of the* | |  | | --- | | Printing all items in a list | |
|  |  |  |
| Getting the first element | *but you can also pop elements from any position in the list.* | for user in users: |
| first\_user = users[0] | |  | | --- | | Pop the last item from a list | | print(user) |
|  |  |  |
|  |
| Getting the second element | |  | | --- | | most\_recent\_user = users.pop() | | |  | | --- | | Printing a message for each item, and a separate message afterwards | |
| second\_user = users[1] | print(most\_recent\_user) |  |
| Getting the last element | |  | | --- | | Pop the first item in a list | | for user in users: |
| print("Welcome, " + user + "!") |
|  |  |  |
| |  | | --- | | newest\_user = users[-1] | | |  | | --- | | first\_user = users.pop(0) | | print("Welcome, we're glad to see you all!") |
| print(first\_user) |
| |  | | --- | |  | | |  | | --- | |  | |  |
| *Once you've defined a list, you can change individual elements in the list. You do this by referring to the index of* |
| |  | | --- | | *The len() function returns the number of items in a list.* | |
| *the item you want to modify.* | |  | | --- | | Find the length of a list | |  |
|  |  |  |
| |  | | --- | | Changing an element | | num\_users = len(users) |  |
| users[0] = 'valerie' | print("We have " + str(num\_users) + " users.") |  |
| users[-2] = 'ronald' |

|  |  |  |  |
| --- | --- | --- | --- |
|  | |  | | --- | |  | |  |
| *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* | *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.* | *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* |
| *large list of numbers.* | |  | | --- | | Making a copy of a list | | *you can't change the individual elements in a tuple.)* |
|  |  |  |
| Printing the numbers 0 to 1000 | |  | | --- | | finishers = ['kai', 'abe', 'ada', 'gus', 'zoe'] | | Defining a tuple |
| fornumer in rnge(101): |  |  |
|  |
| fornumer in rnge(101): | copy\_of\_finishers = finishers[:] | dimensions = (800, 600) |
| print(number) |
|  |  |  |
| Printing the numbers 1 to 1000 | |  | | --- | |  | | Looping through a tuple |
|  |  |  |
| for number in range(1, 1001): | *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* | for dimension in dimensions: |
| print(number) | print(dimension) |
|  |  |  |
| Making a list of numbers from 1 to a million | *comprehensions may look complicated at first; if so, use the* | Overwriting a tuple |
| numbers = list(range(1, 1000001)) | *for loop approach until you're ready to start using comprehensions.* | dimensions = (800, 600) |
| printdimensons |
|  |  | printdimensons |
|  | *values you want to store in the list. Then write a for loop to generate input values needed to make the list.* | dimensions = (1200, 900) |
| *There are a number of simple statistics you can run on a list* |
| *containing numerical data.* | |  | | --- | | Using a loop to generate a list of square numbers | |  |
| Finding the minimum value in a list | |  | | --- | | squares = [] | |  |
| *When you're first learning about data structures such as* |
| ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77] | for x in range(1, 11): | *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* |
| square = x\*\*2 |
| youngest = min(ages) |
| squares.append(square) |
|  |  |  |
| Finding the maximum value | |  | | --- | | Using a comprehension to generate a list of square | | *list. Try running the following code on pythontutor.com, and* |
| ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77] | numbers | *then run your own code.* |
|  |  |  |
| oldest = max(ages) | |  | | --- | | squares = [x\*\*2 for x in range(1, 11)] | | Build a list and print the items in the list |
|  |  |  |
| Finding the sum of all values | |  | | --- | | Using a loop to convert a list of names to upper case | | dogs = [] |
|  |
|  |  | dogs.append('willie') |
| ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77] | |  | | --- | | names = ['kai', 'abe', 'ada', 'gus', 'zoe'] | | dogs.append('hootz') |
| total\_years = sum(ages) |
| dogs.append('peso') |
| |  | | --- | |  | | upper\_names = [] | dogs.append('goblin') |
| for name in names: |
| *You can work with any set of elements from a list. A portion* | upper\_names.append(name.upper()) | for dog in dogs: |
|  |
| *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* | |  | | --- | | Using a comprehension to convert a list of names to upper case | | print("Hello " + dog + "!") |
| print("I love these dogs!") |
| *the beginning of the list, and leave off the last index to slice through the end of the list.* | |  | | --- | | names = ['kai', 'abe', 'ada', 'gus', 'zoe'] | | print("\nThese were my first two dogs:") |
| old\_dogs = dogs[:2] |
|  |
| |  | | --- | | Getting the first three items | | upper\_names = [name.upper() for name in names] | for old\_dog in old\_dogs: |
| print(old\_dog) |
| finishers = ['kai', 'abe', 'ada', 'gus', 'zoe'] | |  | | --- | |  | | del dogs[0] |
| first\_three = finishers[:3] |
| |  | | --- | | *Readability counts* | | dogs.remove('peso') |
| |  | | --- | | Getting the middle three items | |  |  |
| print(dogs) |
|  |  Use four spaces per indentation level. |  |
| middle\_three = finishers[1:4] |  Keep your lines to 79 characters or fewer. |  |
| |  | | --- | | Getting the last three items | |  Use single blank lines to group parts of your | |  | | --- | | *More cheat sheets available at* | |
| last\_three = finishers[-3:] | program visually. |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | |  | | --- | |  | |  |
| *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.* | *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.* |
|  |
|  |
|  | |  | | --- | | Adding a key-value pair | | |  | | --- | | Looping through all key-value pairs | |
| |  | | --- | |  | | |  | | --- | | alien\_0 = {'color': 'green', 'points': 5} | | |  | | --- | | # Store people's favorite languages. | |
| fav\_languages = { |
| 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 | alien\_0['x'] = 0 | 'jen': 'python', |
| 'sarah': 'c', |
| alien\_0['y'] = 25 |
| alien\_0['speed'] = 1.5 | 'edward': 'ruby', |
|  |
| provide a key, Python returns the value associated | |  | | --- | | Adding to an empty dictionary | | 'phil': 'python', |
| } |
| with that key. You can loop through all the key-value pairs, all the keys, or all the values. | |  | | --- | | alien\_0 = {} | | # Show each person's favorite language. |
| alien\_0['color'] = 'green' |
| for name, language in fav\_languages.items(): |
| alien\_0['points'] = 5 |
|  |  | print(name + ": " + language) |
|  | |  | | --- | |  | | |  | | --- | | Looping through all the keys | |
| *Use curly braces to define a dictionary. Use colons to* |
| *connect keys and values, and use commas to separate individual key-value pairs.* | *You can modify the value associated with any key in a dictionary. To do so give the name of the dictionary and* | |  | | --- | | # Show everyone who's taken the survey. | |
| for name in fav\_languages.keys(): |
| Making a dictionary | *enclose the key in square brackets, then provide the new value for that key.* | print(name) |
|  |
|  |  |
|  |  |  |
| alien\_0 = {'color': 'green', 'points': 5} | |  | | --- | | Modifying values in a dictionary | |  |
|  |  |  |
| |  | | --- | |  | | |  | | --- | | alien\_0 = {'color': 'green', 'points': 5} | | |  | | --- | | # Show all the languages that have been chosen. | |
| for language in fav\_languages.values(): |
| print(alien\_0) | print(language) |
| *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* | # Change the alien's color and point value. | |  | | --- | | Looping through all the keys in order | |
| *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* | alien\_0['color'] = 'yellow' | |  | | --- | | # Show each person's favorite language, | |
| alien\_0['points'] = 10 |
| # in order by the person's name. |
| print(alien\_0) |
| for name in sorted(fav\_languages.keys()): |
| *specify a default value to use if the key is not in the* |  | print(name + ": " + language) |
| *dictionary.* |  |
|  |  |
| |  | | --- | | Getting the value associated with a key | | |  | | --- | | *You can remove any key-value pair you want from a*  *dictionary. To do so use the del keyword and the dictionary* | | |  | | --- | |  | |
|  |
| alien\_0 = {'color': 'green', 'points': 5} | *name, followed by the key in square brackets. This will* | |  | | --- | | *You can find the number of key-value pairs in a dictionary.* | |
|  | *delete the key and its associated value.* | |  | | --- | | Finding a dictionary's length | |
| print(alien\_0['color']) | |  | | --- | | Deleting a key-value pair | | num\_responses = len(fav\_languages) |
| print(alien\_0['points']) |
| |  | | --- | | Getting the value with get() | | |  | | --- | | alien\_0 = {'color': 'green', 'points': 5} | |  |
| print(alien\_0) |
| alien\_0 = {'color': 'green'} |  |  |
| alien\_color = alien\_0.get('color') | del alien\_0['points'] |
| print(alien\_0) |
|  |
| alien\_points = alien\_0.get('points', 0) | |  | | --- | |  | |  |
| print(alien\_color) |
| *Try running some of these examples on pythontutor.com.* |
| print(alien\_points) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |  |
| *It's sometimes useful to store a set of dictionaries in a list; this is called nesting.* | *Storing a list inside a dictionary alows you to associate more than one value with each key.* | *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* |
| |  | | --- | | Storing dictionaries in a list | | |  | | --- | | Storing lists in a dictionary | | *preserve the order in which keys and values are added, use* |
| # Start with an empty list. | # Store multiple languages for each person. | *an OrderedDict.* |
| users = [] | fav\_languages = { | Preserving the order of keys and values |
|  |
| # Make a new user, and add them to the list. | 'jen': ['python', 'ruby'], | from collections import OrderedDict |
| 'sarah': ['c'], |
| new\_user = { | 'edward': ['ruby', 'go'], | # Store each person's languages, keeping |
| 'last': 'fermi', | 'phil': ['python', 'haskell'], |
| # track of who respoded first. |
| 'first': 'enrico', | } |
| 'username': 'efermi', | |  | | --- | | # Show all responses for each person. | | fav\_languages = OrderedDict() |
| } |
| users.append(new\_user) | for name, langs in fav\_languages.items(): | fav\_languages['jen'] = ['python', 'ruby'] |
| fav\_languages['sarah'] = ['c'] |
| print(name + ": ") |
| # Make another new user, and add them as well. |
| fav\_languages['edward'] = ['ruby', 'go'] |
| for lang in langs: |
| fav\_languages['phil'] = ['python', 'haskell'] |
| new\_user = { | print("- " + lang) |
|  |
| 'last': 'curie', | |  | | --- | |  | | # Display the results, in the same order they |
| 'first': 'marie', |
| 'username': 'mcurie', | # were entered. |
| *You can store a dictionary inside another dictionary. In this case each value associated with a key is itself a dictionary.* | for name, langs in fav\_languages.items(): |
| } |
| users.append(new\_user) | print(name + ":") |
|  |
| # Show all information about each user. | |  | | --- | | Storing dictionaries in a dictionary | | for lang in langs: |
| print("- " + lang) |
|  |  |  |
| for user\_dict in users: | users = { |  |
| 'aeinstein': { |
| for k, v in user\_dict.items(): |
| print(k + ": " + v) | 'first': 'albert', |
| *You can use a loop to generate a large number of*  *dictionaries efficiently, if all the dictionaries start out with* |
| 'last': 'einstein', |
| print("\n") |
|  |
| |  | | --- | | You can also define a list of dictionaries directly, | | 'location': 'princeton', | *similar data.* |
| }, |
| without using append(): | 'mcurie': { | A million aliens |
| # Define a list of users, where each user | 'first': 'marie', | aliens = [] |
| 'last': 'curie', |
| # is represented by a dictionary. |
| 'location': 'paris', | # Make a million green aliens, worth 5 points |
| users = [ |
| }, |
| { |
| } | # each. Have them all start in one row. |
| 'last': 'fermi', |
| for username, user\_dict in users.items(): | for alien\_num in range(1000000): |
| 'first': 'enrico', |
| new\_alien = {} |
| 'username': 'efermi', |
| print("\nUsername: " + username) | new\_alien['color'] = 'green' |
| }, |
| new\_alien['points'] = 5 |
| full\_name = user\_dict['first'] + " " |
| { |
| full\_name += user\_dict['last'] | new\_alien['x'] = 20 \* alien\_num |
| 'last': 'curie', |
| location = user\_dict['location'] | new\_alien['y'] = 0 |
| 'first': 'marie', |
| |  | | --- | | print("\tFull name: " + full\_name.title()) | | aliens.append(new\_alien) |
| 'username': 'mcurie', |
| }, | # Prove the list contains a million aliens. |
| print("\tLocation: " + location.title()) |
| ] |
| nm\_alien= n(alien) |
|  |  | nm\_alien= n(alien) |
| # Show all information about each user. | |  | | --- | |  | | print("Number of aliens created:") |
| for user\_dict in users: | *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* | print(num\_aliens) |
| for k, v in user\_dict.items(): |
| print(k + ": " + v) |
| print("\n") | *are probably simpler ways of managing your data, such as using classes.* | |  | | --- | | *More cheat sheets available at* | |
|  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| *Testing numerical values is similar to testing string values.* | *Several kinds of if statements exist. Your choice of which to* |
|  | |  | | --- | | Testing equality and inequality | | *use depends on the number of conditions you need to test. You can have as many elif blocks as you need, and the* |
|  |
|  | |  | | --- | | >>> age = 18 | | *else block is always optional.* |
|  | >>> age == 18 | |  | | --- | | Simple if statement | |
|  | True |
|  |  |
| >>> age != 18 | |  | | --- | | age = 19 | |
| False |  |
|  | |  | | --- | | Comparison operators | | if age >= 18: |
| print("You're old enough to vote!") |
|  | |  | | --- | | >>> age = 19 | | |  | | --- | | If-else statements | |
| >>> age < 21 |
|  |  |  |
| |  | | --- | |  | | True | |  | | --- | | age = 17 | |
| >>> age <= 21 | if age >= 18: |
| True |
| 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 | >>> age > 21 | print("You're old enough to vote!") |
| False | else: |
| >>> age >= 21 | print("You can't vote yet.") |
|  | |  | | --- | | The if-elif-else chain | |
| condition, or you can create a complex series of if | False |
|  |  |  |
| statements that idenitfy the exact conditions you're looking for. | |  | | --- | |  | | |  | | --- | | age = 12 | |
| *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.* | if age < 4: |
| While loops run as long as certain conditions remain true. You can use while loops to let your programs |
| price = 0 |
| elif age < 18: |
| run as long as your users want them to. | |  | | --- | | Using and to check multiple conditions | | price = 5 |
| else: |
|  | |  | | --- | | >>> age\_0 = 22 | |  |
| price = 10 |
| >>> age\_1 = 18 | print("Your cost is $" + str(price) + ".") |
| *A conditional test is an expression that can be evaluated as True or False. Python uses the values True and False to* |
| >>> age\_0 >= 21 and age\_1 >= 21 |
|  |
|  | False |
| |  | | --- | |  | |
| *decide whether the code in an if statement should be executed.* | >>> age\_1 = 23 |
| >>> age\_0 >= 21 and age\_1 >= 21 |
|  |
| Checking for equality | True | |  | | --- | | *You can easily test whether a certain value is in a list. You* | |
| *A single equal sign assigns a value to a variable. A double equal sign (==) checks whether two values are equal.* | |  | | --- | | Using or to check multiple conditions | | *can also test whether a list is empty before trying to loop through the list.* |
|  |  |  |
| >>> car = 'bmw' | |  | | --- | | >>> age\_0 = 22 | | |  | | --- | | Testing if a value is in a list | |
| >>> age\_1 = 18 |
| >>> car == 'bmw' |  |
| >>> age\_0 >= 21 or age\_1 >= 21 | >>> players = ['al', 'bea', 'cyn', 'dale'] |
| True |
| True | >>> 'al' in players |
| >>> car = 'audi' |
| >>> age\_0 = 18 | True |
| >>> car == 'bmw' |
| >>> age\_0 >= 21 or age\_1 >= 21 | >>> 'eric' in players |
| False |
| False | False |
|  |  |  |
| Ignoring case when making a comparison |  |  |
| >>> car = 'Audi' | |  | | --- | |  | |  |
| >>> car.lower() == 'audi' | *A boolean value is either True or False. Variables with boolean values are often used to keep track of certain* |
| True |
| Checking for inequality | *conditions within a program.* |  |
|  |  |  |
| >>> topping = 'mushrooms' | |  | | --- | | Simple boolean values | |  |
|  | |  | | --- | | game\_active = True | |  |
| >>> topping != 'anchovies' |
| True | can\_edit = False |

|  |  |  |  |
| --- | --- | --- | --- |
|  | |  | | --- | |  | |  |
|  |  |  |
| Testing if a value is not in a list | |  | | --- | | Letting the user choose when to quit | | Using continue in a loop |
| banned\_users = ['ann', 'chad', 'dee'] | |  | | --- | | prompt = "\nTell me something, and I'll " | | banned\_users = ['eve', 'fred', 'gary', 'helen'] |
| user = 'erin' | prompt += "repeat it back to you." | prompt = "\nAdd a player to your team." |
| prompt += "\nEnter 'quit' to end the program. " |
| if user not in banned\_users: | message = "" | prompt += "\nEnter 'quit' when you're done. " |
| print("You can play!") |
|  |  |
| Checking if a list is empty | while message != 'quit': | players = [] |
| message = input(prompt) | while True: |
| players = [] | if message != 'quit': | player = input(prompt) |
| if player == 'quit': |
| if players: | print(message) | break |
|  |
| for player in players: | |  | | --- | | Using a flag | | elif player in banned\_users: |
| print(player + " is banned!") |
| print("Player: " + player.title()) |
|  | |  | | --- | | prompt = "\nTell me something, and I'll " | | continue |
| else: |
| else: |
| print("We have no players yet!") | prompt += "repeat it back to you." |
| players.append(player) |
|  |
|  | prompt += "\nEnter 'quit' to end the program. " |  |
|  | active = True | print("\nYour team:") |
| *You can allow your users to enter input using the input() statement. In Python 3, all input is stored as a string.* | for player in players: |
| while active: |
| print(player) |
| message = input(prompt) |
| Simple input | if message == 'quit': |  |
| name = input("What's your name? ") |  |  |
| active = False | *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* |
| print("Hello, " + name + ".") | else: |
|  |
| Accepting numerical input | print(message) | *become False, the loop will never stop running.* |
|  |  |  |
| age = input("How old are you? ") | |  | | --- | | Using break to exit a loop | | An infinite loop |
|  |  |  |
| age = int(age) | |  | | --- | | prompt = "\nWhat cities have you visited?" | | while True: |
| if age >= 18: | prompt += "\nEnter 'quit' when you're done. " | name = input("\nWho are you? ") |
| while True: | print("Nice to meet you, " + name + "!") |
| print("\nYou can vote!") |
|  |  |  |
| else: | city = input(prompt) |  |
| 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.* | if city == 'quit': | *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.* |
| break |
| else: |
| print("I've been to " + city + "!") |
|  |  |  |
| name = raw\_input("What's your name? ") | |  | | --- | |  | | Removing all cats from a list of pets |
| print("Hello, " + name + ".") |
| |  | | --- | |  | | *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.* | pets = ['dog', 'cat', 'dog', 'fish', 'cat', |
| 'rabbit', 'cat'] |
| print(pets) |
| *A while loop repeats a block of code as long as a condition is True.* |
| while 'cat' in pets: |
|  |  |  |
| |  | | --- | | Counting to 5 | |  | pets.remove('cat') |
| current\_number = 1 |  | print(pets) |
| |  | | --- | | *You can use the break statement and the continue*  *statement with any of Python's loops. For example you can* | |
| while current\_number <= 5: | *use break to quit a for loop that's working through a list or a* |  |
| print(current\_number) | *dictionary. You can use continue to skip over certain items when looping through a list or dictionary as well.* | |  | | --- | | *More cheat sheets available at* | |
| current\_number += 1 |
|  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | |  | | --- | |  | | |  | | --- | |  | |
| *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.* | *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.* |
|  |
|  | *With keyword arguments, you specify which parameter each argument should be assigned to in the function call.* | |  | | --- | | Returning a single value | |
|  |
|  | *When you use keyword arguments, the order of the arguments doesn't matter.* | |  | | --- | | def get\_full\_name(first, last): | |
| """Return a neatly formatted full name.""" |
|  |  |  |
|  | |  | | --- | | Using positional arguments | | full\_name = first + ' ' + last |
| return full\_name.title() |
|  | |  | | --- | | def describe\_pet(animal, name): | |  |
| Functions are named blocks of code designed to do one specific job. Functions allow you to write code | """Display information about a pet.""" | musician = get\_full\_name('jimi', 'hendrix') |
| print("\nI have a " + animal + ".") | print(musician) |
|  | |  | | --- | | Returning a dictionary | |
| once that can then be run whenever you need to | print("Its name is " + name + ".") |
| accomplish the same task. Functions can take in the information they need, and return the information they generate. Using functions effectively makes your | describe\_pet('hamster', 'harry') | |  | | --- | | def build\_person(first, last): | |
| describe\_pet('dog', 'willie') | """Return a dictionary of information |
| programs easier to write, read, test, and fix. | |  | | --- | | Using keyword arguments | | about a person. |
| """ |
|  |  |  |
|  |  |  |
|  | def describe\_pet(animal, name): | person = {'first': first, 'last': last} |
| |  | | --- | | """Display information about a pet.""" | | return person |
| *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,* | print("\nI have a " + animal + ".") | musician = build\_person('jimi', 'hendrix') |
| print("Its name is " + name + ".") |
|  | print(musician) |
| *describes what the function does. The body of a function is indented one level.* | describe\_pet(animal='hamster', name='harry') | |  | | --- | | Returning a dictionary with optional values | |
| describe\_pet(name='willie', animal='dog') |
| *To call a function, give the name of the function followed by a set of parentheses.* | |  | | --- | |  | | def build\_person(first, last, age=None): |
| """Return a dictionary of information |
|  |
| Making a function | *You can provide a default value for a parameter. When function calls omit this argument the default value will be* | about a person. |
| """ |
| def greet\_user(): | *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.* | person = {'first': first, 'last': last} |
| if age: |
| """Display a simple greeting.""" |
| print("Hello!") | person['age'] = age |
| return person |
| greet\_user() | |  | | --- | | Using a default value | |  |
|  |
|  |  | muicia = uild\_prson('jii','hendrx',27) |
| |  | | --- | |  | | def describe\_pet(name, animal='dog'): | |  | | --- | | print(musician) | |
| |  | | --- | | """Display information about a pet.""" | |
| *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* | print("\nI have a " + animal + ".") | musician = build\_person('janis', 'joplin') |
| print("Its name is " + name + ".") |
| print(musician) |
| *the function's name, and parameters are listed in parentheses in the function's definition.* | describe\_pet('harry', 'hamster') |
| |  | | --- | |  | |
| describe\_pet('willie') |
|  |
| |  | | --- | | Passing a single argument | | |  | | --- | | Using None to make an argument optional | | *Try running some of these examples on pythontutor.com.* |
| def greet\_user(username): | |  | | --- | | def describe\_pet(animal, name=None): | | |  |  |  |  | | --- | --- | --- | --- | | |  |  |  | | --- | --- | --- | | *Covers Python 3 and Python 2* | |  | | --- | |  | | | |
| """Display a simple greeting.""" | """Display information about a pet.""" |
| print("Hello, " + username + "!") | print("\nI have a " + animal + ".") |
| greet\_user('jesse') | if name: |
| print("Its name is " + name + ".") |
| greet\_user('diana') | describe\_pet('hamster', 'harry') |
| greet\_user('brandon') |
| describe\_pet('snake') |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | |
| *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.* | *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.* | *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.)* |
| |  | | --- | | Passing a list as an argument | | *The \*\* operator allows a parameter to collect an arbitrary number of keyword arguments.* | |  | | --- | | Storing a function in a module | |
| def greet\_users(names): |  | *File: pizza.py* |
|  | |  | | --- | | Collecting an arbitrary number of arguments | | def make\_pizza(size, \*toppings): |
| """Print a simple greeting to everyone.""" |
|  | |  | | --- | | def make\_pizza(size, \*toppings): | | """Make a pizza.""" |
| for name in names: |
| print("\nMaking a " + size + " pizza.") |
| msg = "Hello, " + name + "!" | """Make a pizza.""" |
| print(msg) | print("\nMaking a " + size + " pizza.") | print("Toppings:") |
| for topping in toppings: |
| print("Toppings:") |
| usernames = ['hannah', 'ty', 'margot'] |
| for topping in toppings: | print("- " + topping) |
|  |
| greet\_users(usernames) | print("- " + topping) | |  | | --- | | Importing an entire module | |
| |  | | --- | | Allowing a function to modify a list  *The following example sends a list of models to a function for* | | # Make three pizzas with different toppings. | *File: making\_pizzas.py*  *Every function in the module is available in the program file.* |
| make\_pizza('small', 'pepperoni') |
| *printing. The original list is emptied, and the second list is filled.* |  | import pizza |
| make\_pizza('large', 'bacon bits', 'pineapple') |
| def print\_models(unprinted, printed): |  | pizza.make\_pizza('medium', 'pepperoni') |
| make\_pizza('medium', 'mushrooms', 'peppers', |
| """3d print a set of models.""" | 'onions', 'extra cheese') |
|  |  |
| while unprinted: | |  | | --- | | Collecting an arbitrary number of keyword arguments | | pizza.make\_pizza('small', 'bacon', 'pineapple') |
| current\_model = unprinted.pop() |  | |  | | --- | | Importing a specific function | |
| print("Printing " + current\_model) | def build\_profile(first, last, \*\*user\_info): | *Only the imported functions are available in the program file.* |
| printed.append(current\_model) |  |
|  | """Build a user's profile dictionary.""" | from pizza import make\_pizza |
| # Build a dict with the required keys. |
| # Store some unprinted designs, | profile = {'first': first, 'last': last} | make\_pizza('medium', 'pepperoni') |
| # and print each of them. |
| unprinted = ['phone case', 'pendant', 'ring'] | # Add any other keys and values. | make\_pizza('small', 'bacon', 'pineapple') |
|  |
| printed = [] | for key, value in user\_info.items(): | |  | | --- | | Giving a module an alias | |
| print\_models(unprinted, printed) | profile[key] = value |

|  |  |  |
| --- | --- | --- |
| print("\nUnprinted:", unprinted) | return profile | import pizza as p |
| print("Printed:", printed) | p.make\_pizza('medium', 'pepperoni') |
| |  | | --- | | Preventing a function from modifying a list | | # Create two users with different kinds | p.make\_pizza('small', 'bacon', 'pineapple') |
| *The following example is the same as the previous one, except the original list is unchanged after calling print\_models().* | # of information. | Giving a function an alias |
| user\_0 = build\_profile('albert', 'einstein', |
|  |  |  |
| def print\_models(unprinted, printed): | location='princeton') | from pizza import make\_pizza as mp |
| user\_1 = build\_profile('marie', 'curie', |
| """3d print a set of models.""" | location='paris', field='chemistry') | mp('medium', 'pepperoni') |
| while unprinted: |
| mp('small', 'bacon', 'pineapple') |
|  |
| current\_model = unprinted.pop() |
|  | print(user\_0) | Importing all functions from a module  *Don't do this, but recognize it when you see it in others' code. It* |
| print("Printing " + current\_model) |
| print(user\_1) |
| printed.append(current\_model) |
|  |  | *can result in naming conflicts, which can cause errors.* |
| # Store some unprinted designs, |  | from pizza import \* |
| |  | | --- | | *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* | |
| # and print each of them. |
| original = ['phone case', 'pendant', 'ring'] | make\_pizza('medium', 'pepperoni') |
| printed = [] | make\_pizza('small', 'bacon', 'pineapple') |
|  |
| print\_models(original[:], printed) | *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.* | |  | | --- | | *More cheat sheets available at* | |
| print("\nOriginal:", original) |
| print("Printed:", printed) |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | |  | | --- | |  | | |  | | --- | |  | |
|  |
|  | |  | | --- | | Creating an object from a class | | *If the class you're writing is a specialized version of another class, you can use inheritance. When one class inherits* |
|  | |  | | --- | | my\_car = Car('audi', 'a4', 2016) | | *from another, it automatically takes on all the attributes and methods of the parent class. The child class is free to* |
|  | |  | | --- | | Accessing attribute values | | *introduce new attributes and methods, and override* |
| |  | | --- | |  | | |  | | --- | | print(my\_car.make) | | *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.* |
| print(my\_car.model) |
| print(my\_car.year) |
|  |  |  |
| Classes are the foundation of object-oriented | |  | | --- | | Calling methods | | |  | | --- | | The \_\_init\_\_() method for a child class | |
|  |  |  |
| you want to model in your programs: for example dogs, cars, and robots. You use a class to make | |  | | --- | | my\_car.fill\_tank() | | class ElectricCar(Car): |
| """A simple model of an electric car.""" |
| my\_car.drive() |
| objects, which are specific instances of dogs, cars, and robots. A class defines the general behavior that | |  | | --- | | Creating multiple objects | | def \_\_init\_\_(self, make, model, year): |
|  |
| 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 | |  | | --- | | my\_car = Car('audi', 'a4', 2016) | | """Initialize an electric car.""" |
| super().\_\_init\_\_(make, model, year) |
| my\_old\_car = Car('subaru', 'outback', 2013) |
| my\_truck = Car('toyota', 'tacoma', 2010) | # Attributes specific to electric cars. |
| write a class that extends the functionality of an  existing class. This allows you to code efficiently for a wide variety of situations. | |  | | --- | |  | | # Battery capacity in kWh. |
| self.battery\_size = 70 |
| *You can modify an attribute's value directly, or you can* | # Charge level in %. |
| self.charge\_level = 0 |
|  |  |  |
|  |  |  |
| |  | | --- | |  | | |  | | --- | | Modifying an attribute directly | | |  | | --- | | Adding new methods to the child class | |
|  |  |
| *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* | my\_new\_car = Car('audi', 'a4', 2016) | class ElectricCar(Car): |
| --snip-- |
| my\_new\_car.fuel\_level = 5 |
| def charge(self): |
| *attributes, and the behavior is represented by functions. Functions that are part of a class are called methods.* | |  | | --- | | Writing a method to update an attribute's value | |  |
| """Fully charge the vehicle.""" |
|  |
|  | def update\_fuel\_level(self, new\_level): | self.charge\_level = 100 |
| |  | | --- | | The Car class | |  | print("The vehicle is fully charged.") |
| """Update the fuel level.""" |
|  |  |  |
|  |
| class Car(): | if new\_level <= self.fuel\_capacity: | |  | | --- | | Using child methods and parent methods | |
|  | my\_ecar = ElectricCar('tesla', 'model s', 2016) |
| """A simple attempt to model a car.""" | self.fuel\_level = new\_level |
| else: |
| def \_\_init\_\_(self, make, model, year): | print("The tank can't hold that much!") | my\_ecar.charge() |
| """Initialize car attributes.""" | |  | | --- | | Writing a method to increment an attribute's value | | my\_ecar.drive() |
|  |
| self.make = make | def add\_fuel(self, amount): | |  | | --- | |  | |
| self.model = model |
| self.year = year | """Add fuel to the tank.""" |
| if (self.fuel\_level + amount |
| # Fuel capacity and level in gallons. | <= self.fuel\_capacity): | *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.* |
| self.fuel\_capacity = 15 | self.fuel\_level += amount |
| print("Added fuel.") |
| self.fuel\_level = 0 |
| else: |
|  | print("The tank won't hold that much.") |  |
| def fill\_tank(self): |  |
| """Fill gas tank to capacity.""" |
| self.fuel\_level = self.fuel\_capacity | |  | | --- | |  | | |  |  |  |  | | --- | --- | --- | --- | | |  |  |  | | --- | --- | --- | | *Covers Python 3 and Python 2* | |  | | --- | |  | | | |
| print("Fuel tank is full.") |
| def drive(self): | *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.* |
| """Simulate driving.""" |
| print("The car is moving.") |

|  |  |  |  |
| --- | --- | --- | --- |
|  | |  | | --- | |  | |  |
|  |
| Overriding parent methods | *Class files can get long as you add detailed information and functionality. To help keep your program files uncluttered,* | Classes should inherit from object |
| class ElectricCar(Car): | *you can store your classes in modules and import the classes you need into your main program.* | class ClassName(object): |
|  |
| --snip-- |  | The Car class in Python 2.7 |
| def fill\_tank(self): | |  | | --- | | Storing classes in a file | |  |
| """Display an error message.""" | *car.py* | class Car(object): |
|  |
| print("This car has no fuel tank!") |  | Child class \_\_init\_\_() method is different |
|  | """Represent gas and electric cars.""" |  |
|  | class Car(): | class ChildClassName(ParentClass): |
| def \_\_init\_\_(self): |
| *A class can have objects as attributes. This allows classes to work together to model complex situations.* | """A simple attempt to model a car.""" |
| super(ClassName, self).\_\_init\_\_() |
| --snip— |
|  |  |  |
| A Battery class |  | The ElectricCar class in Python 2.7 |
|  | class Battery(): |  |
| class Battery(): | """A battery for an electric car.""" | class ElectricCar(Car): |
| def \_\_init\_\_(self, make, model, year): |
| --snip-- |
| """A battery for an electric car.""" | super(ElectricCar, self).\_\_init\_\_( |
| def \_\_init\_\_(self, size=70): | class ElectricCar(Car): | make, model, year) |
|  |  |  |
| """Initialize battery attributes.""" | """A simple model of an electric car.""" |  |
| --snip-- |
| # Capacity in kWh, charge level in %. |
|  |
| self.size = size | |  | | --- | | Importing individual classes from a module *my\_cars.py* | | *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.* |
| self.charge\_level = 0 |
| def get\_range(self): | from car import Car, ElectricCar | *Here's an example showing how to make a fleet of rental cars, and make sure all the cars are ready to drive.* |
| """Return the battery's range.""" |  |
| if self.size == 70: | my\_beetle = Car('volkswagen', 'beetle', 2016) | A fleet of rental cars |
| my\_beetle.fill\_tank() |
| return 240 |
| elif self.size == 85: | my\_beetle.drive() | from car import Car, ElectricCar |
| return 270 | my\_tesla = ElectricCar('tesla', 'model s', |  |
| Using an instance as an attribute |  | # Make lists to hold a fleet of cars. |
| 2016) |
| gas\_fleet = [] |
|  |
| class ElectricCar(Car): | my\_tesla.charge() | electric\_fleet = [] |
| my\_tesla.drive() |
| --snip-- |
|  | |  | | --- | | Importing an entire module | | # Make 500 gas cars and 250 electric cars. |
|  | for \_ in range(500): |
| def \_\_init\_\_(self, make, model, year): | import car |  |
| car = Car('ford', 'focus', 2016) |
| """Initialize an electric car.""" |
| my\_beetle = car.Car( | gas\_fleet.append(car) |
| super().\_\_init\_\_(make, model, year) |
| for \_ in range(250): |
| # Attribute specific to electric cars. | 'volkswagen', 'beetle', 2016) | ecar = ElectricCar('nissan', 'leaf', 2016) |
| my\_beetle.fill\_tank() | electric\_fleet.append(ecar) |
| self.battery = Battery() |
| my\_beetle.drive() |
| def charge(self): | my\_tesla = car.ElectricCar( | # Fill the gas cars, and charge electric cars. |
| """Fully charge the vehicle.""" | for car in gas\_fleet: |
| 'tesla', 'model s', 2016) | car.fill\_tank() |
| self.battery.charge\_level = 100 |
| my\_tesla.charge() | for ecar in electric\_fleet: |
| print("The vehicle is fully charged.") |
| my\_tesla.drive() |  |
|  |  | ecar.charge() |
| Using the instance | |  | | --- | | Importing all classes from a module | |  |
| my\_ecar = ElectricCar('tesla', 'model x', 2016) | *(Don’t do this, but recognize it when you see it.)* | print("Gas cars:", len(gas\_fleet)) |
| print("Electric cars:", len(electric\_fleet)) |
|  |  |  |
|  | from car import \* |  |
| my\_ecar.charge() | my\_beetle = Car('volkswagen', 'beetle', 2016) | *More cheat sheets available at* |
| print(my\_ecar.battery.get\_range()) |
| my\_ecar.drive() |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | |  | | --- | |  | |  |
|  | |  | | --- | | Storing the lines in a list | | Opening a file using an absolute path |
|  | |  | | --- | | filename = 'siddhartha.txt' | | f\_path = "/home/ehmatthes/books/alice.txt" |
| with open(filename) as f\_obj: | with open(f\_path) as f\_obj: |
|  | lines = f\_obj.readlines() | lines = f\_obj.readlines() |
| Opening a file on Windows  *Windows will sometimes interpret forward slashes incorrectly. If* |
|  | for line in lines: |
|  |
|  | print(line.rstrip()) | *you run into this, use backslashes in your file paths.* |
| f\_path = "C:\Users\ehmatthes\books\alice.txt" |
|  |  |
| 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 |  | with open(f\_path) as f\_obj: |
| |  | | --- | | *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* | |
| lines = f\_obj.readlines() |
| to files allows users to pick up where they left off the | *Python you want to append to the end of an existing file.* |  |
| next time they run your program. You can write text to files, and you can store Python structures such as | |  | | --- | | Writing to an empty file | |  |
| *When you think an error may occur, you can write a try-* |
| lists in data files. | |  | | --- | | filename = 'programming.txt' | | *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.* |
| Exceptions are special objects that help your  programs respond to errors in appropriate ways. For | with open(filename, 'w') as f: |
| f.write("I love programming!") |
|  |  |  |
| example if your program tries to open a file that | |  | | --- | | Writing multiple lines to an empty file | | |  | | --- | | Handling the ZeroDivisionError exception | |
| doesn’t exist, you can use exceptions to display an |
| informative error message instead of having the program crash. | |  | | --- | | filename = 'programming.txt' | | try: |
| print(5/0) |
| with open(filename, 'w') as f: | except ZeroDivisionError: |
|  | f.write("I love programming!\n") | print("You can't divide by zero!") |
| |  | | --- | | Handling the FileNotFoundError exception | |
|  | f.write("I love creating new games.\n") |
|  |
| |  | | --- | | *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* | | |  | | --- | | Appending to a file | | f\_name = 'siddhartha.txt' |
| *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.* | |  | | --- | | filename = 'programming.txt' | | try: |
| with open(filename, 'a') as f: | with open(f\_name) as f\_obj: |
|  |  |
| |  | | --- | | Reading an entire file at once | | f.write("I also love working with data.\n") |  |
| except FileNotFoundError: |
| f.write("I love making apps as well.\n") |
| filename = 'siddhartha.txt' |  | msg = "Can't find file {0}.".format(f\_name) |
|  |  | print(msg) |
| with open(filename) as f\_obj: |  |
| |  | | --- | | *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* | |  |
| contents = f\_obj.read() |
| print(contents) | *It can be hard to know what kind of exception to handle when writing code. Try writing your code without a try block,* |
| |  | | --- | | Reading line by line | | |  | | --- | | *any file on your system.* | | *and make it generate an error. The traceback will tell you what kind of exception your program needs to handle.* |
| *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* | |  | | --- | | Opening a file from a subfolder | |  |
| *character. The rstrip() method gets rid of the the extra blank lines this would result in when printing to the terminal.* | f\_path = "text\_files/alice.txt" |  |
| filename = 'siddhartha.txt' | with open(f\_path) as f\_obj: |  |
| with open(filename) as f\_obj: | lines = f\_obj.readlines() |
| for line in f\_obj: | for line in lines: |
| print(line.rstrip()) | print(line.rstrip()) |

|  |  |  |  |
| --- | --- | --- | --- |
|  | |  | | --- | |  | |  |
| *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.* | *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* | *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* |
| Using an else block | *you to do this.* | *specific to Python, so you can share this kind of data with* |
|  |  | *people who work in other languages as well.* |
| print("Enter two numbers. I'll divide them.") | |  | | --- | | Using the pass statement in an else block | |  |
|  | f\_names = ['alice.txt', 'siddhartha.txt', | *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.* |
| x = input("First number: ") | |  | | --- | | 'moby\_dick.txt', 'little\_women.txt'] | |
| y = input("Second number: ") |
| try: | for f\_name in f\_names: | Using json.dump() to store data |
| # Report the length of each file found. |
|  |  | """Store some numbers.""" |
| result = int(x) / int(y) | try: |
| except ZeroDivisionError: | with open(f\_name) as f\_obj: | import json |
| print("You can't divide by zero!") | lines = f\_obj.readlines() |
| else: | except FileNotFoundError: | numbers = [2, 3, 5, 7, 11, 13] |
| print(result) | # Just move on to the next file. |
|  |
| 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.* | pass | filename = 'numbers.json' |
| else: |
| num\_lines = len(lines) | with open(filename, 'w') as f\_obj: |
| msg = "{0} has {1} lines.".format( | json.dump(numbers, f\_obj) |
|  |  |  |
| """A simple calculator for division only.""" |  | Using json.load() to read data |
| print(msg) |
|  |  |  |
| print("Enter two numbers. I'll divide them.") | |  | | --- | |  | | """Load some previously stored numbers.""" |
| print("Enter 'q' to quit.") | import json |
| *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.* |
| while True: | filename = 'numbers.json' |
| x = input("\nFirst number: ") | with open(filename) as f\_obj: |
| if x == 'q': |
| numbers = json.load(f\_obj) |
| break |
| y = input("Second number: ") | *If you want to use a try block and you're not sure which* | print(numbers) |
| if y == 'q': |
|  | *exception to catch, use Exception. It will catch most exceptions, but still allow you to interrupt programs* | Making sure the stored data exists |
| break |
| try: | *intentionally.* | import json |
| result = int(x) / int(y) | |  | | --- | | Don’t use bare except blocks | | f\_name = 'numbers.json' |
| except ZeroDivisionError: |
|  | |  | | --- | | try: | | try: |
| print("You can't divide by zero!") |
| else: | # Do something |
| with open(f\_name) as f\_obj: |
| print(result) | except: |
| numbers = json.load(f\_obj) |
|  |
|  | pass | except FileNotFoundError: |
|  | |  | | --- | | Use Exception instead | |  |
| msg = "Can’t find {0}.".format(f\_name) |
|  |  |
| *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* | |  | | --- | | try: | | print(msg) |
| else: |
| # Do something |
| print(numbers) |
| except Exception: |
|  |  |  |
| *an exception being raised.* | pass | Practice with exceptions |
|  | |  | | --- | | Printing the exception | | *Take a program you've already written that prompts for user input, and add some error-handling code to the program.* |
|  |
| *It's up to you how to communicate errors to your users. Sometimes users need to know if a file is missing;* | |  | | --- | | try: | |  |
| *sometimes it's better to handle the error silently. A little experience will help you know how much to report.* | # Do something | |  | | --- | | *More cheat sheets available at* | |
| except Exception as e: |
|  |
| print(e, type(e)) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | |  | | --- | |  | | |  | | --- | |  | |
|  | |  | | --- | | 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* | | |  | | --- | | 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.* |
|  | import unittest | E |
|  |
| from full\_names import get\_full\_name | ================================================ |
| |  | | --- | | class NamesTestCase(unittest.TestCase): | | ERROR: test\_first\_last (\_\_main\_\_.NamesTestCase) |
| Test names like Janis Joplin. |
|  |  |  |
| |  | | --- | |  | | """Tests for names.py.""" | ------------------------------------------------ |
|  | Traceback (most recent call last): |
|  |
| 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 | def test\_first\_last(self): | File "test\_full\_names.py", line 10, |
| """Test names like Janis Joplin.""" | in test\_first\_last |
| full\_name = get\_full\_name('janis', | 'joplin') |
| 'joplin') | TypeError: get\_full\_name() missing 1 required |
| self.assertEqual(full\_name, | positional argument: 'last' |
| 'Janis Joplin') | ------------------------------------------------ |
| unittest.main() | Ran 1 test in 0.001s |
| also add new features to your programs and know that you haven't broken existing behavior. | |  | | --- | | Running the test  *Python reports on each unit test in the test case. The dot reports a* | | FAILED (errors=1) |
| A unit test verifies that one specific aspect of your code works as it's supposed to. A test case is 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.* | |  | | --- | | 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* | |
| collection of unit tests which verify your code's | |  | | --- | | . | | *your new code.) Here we can make the middle name optional.* |
| behavior in a wide variety of situations. | --------------------------------------- | def get\_full\_name(first, last, middle=''): |
| Ran 1 test in 0.000s |  |
|  |  |  |
| |  | | --- | |  | | OK |  |
| if middle: |
| full\_name = "{0} {1} {2}".format(first, |
| *Python's unittest module provides tools for testing your* |
| *code. To try it out, we’ll create a function that returns a full* | |  | | --- | |  | | middle, last) |
| else: |
| *name. We’ll use the function in a regular program, and then build a test case for the function.* |
| *Failing tests are important; they tell you that a change in the code has affected existing behavior. When a test fails, you* | full\_name = "{0} {1}".format(first, |
| last) |
| |  | | --- | | A function to test | | *need to modify the code so the existing behavior still works.* | return full\_name.title() |
|  |  |  |
|  | |  | | --- | | Modifying the function | | |  | | --- | | Running the test | |
| def get\_full\_name(first, last): | |  | | --- | | *We’ll modify* get\_full\_name() *so it handles middle names, but we’ll do it in a way that breaks existing behavior.* | | *Now the test should pass again, which means our original functionality is still intact.* |
| """Return a full name.""" |
|  |  |  |
| full\_name = "{0} {1}".format(first, last) | def get\_full\_name(first, middle, last): | . |
| return full\_name.title() | --------------------------------------- |
| """Return a full name.""" |
| |  | | --- | | Using the function  *Save this as names.py* | |  | Ran 1 test in 0.000s |
| full\_name = "{0} {1} {2}".format(first, |
| middle, last) | OK |
| from full\_names import get\_full\_name | return full\_name.title() |  |
|  |  |  |
|  | |  | | --- | | Using the function | |  |
| janis = get\_full\_name('janis', 'joplin') | from full\_names import get\_full\_name | |  |  |  |  | | --- | --- | --- | --- | | |  |  |  | | --- | --- | --- | | *Covers Python 3 and Python 2* | |  | | --- | |  | | | |
| print(janis) |
| bob = get\_full\_name('bob', 'dylan') | john = get\_full\_name('john', 'lee', 'hooker') |
| print(bob) | print(john) |
| david = get\_full\_name('david', 'lee', 'roth') |
| print(david) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | |  | | --- | |  | | |  | | --- | |  | |
| *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* | *Testing a class is similar to testing a function, since you’ll mostly be testing your methods.* | *When testing a class, you usually have to make an instance of the class. The* setUp() *method is run before every test.* |
| *class.* | |  | | --- | | A class to test | | *Any instances you make in* setUp() *are available in every* |
| Testing middle names | *Save as accountant.py* | *test you write.* |
|  |  |  |
| *We’ve shown that* get\_full\_name() *works for first and last names. Let’s test that it works for middle names as well.* | class Accountant(): | |  | | --- | | Using setUp() to support multiple tests  *The instance* self.acc *can be used in each new test.* | |
| """Manage a bank account.""" |
| import unittest |  | import unittest |
| from full\_names import get\_full\_name | def \_\_init\_\_(self, balance=0): | from accountant import Accountant |

|  |  |  |
| --- | --- | --- |
| class NamesTestCase(unittest.TestCase): | self.balance = balance | class TestAccountant(unittest.TestCase): |
| """Tests for names.py.""" | def deposit(self, amount): | """Tests for the class Accountant.""" |

|  |  |  |
| --- | --- | --- |
| def test\_first\_last(self): | self.balance += amount | def setUp(self): |
| """Test names like Janis Joplin.""" | def withdraw(self, amount): | self.acc = Accountant() |
| full\_name = get\_full\_name('janis', | self.balance -= amount |
|  |
|  |
| 'joplin') | |  | | --- | | 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.* | | def test\_initial\_balance(self): |
| self.assertEqual(full\_name, | # Default balance should be 0. |
| 'Janis Joplin') | self.assertEqual(self.acc.balance, 0) |
| def test\_middle(self): | import unittest | # Test non-default balance. |
| """Test names like David Lee Roth.""" | from accountant import Accountant |
| acc = Accountant(100) |
| full\_name = get\_full\_name('david', | class TestAccountant(unittest.TestCase): | self.assertEqual(acc.balance, 100) |
| 'roth', 'lee') |
| self.assertEqual(full\_name, | """Tests for the class Accountant.""" | def test\_deposit(self): |
| 'David Lee Roth') | def test\_initial\_balance(self): |
| # Test single deposit. |
| self.acc.deposit(100) |
| unittest.main() | # Default balance should be 0. |
| self.assertEqual(self.acc.balance, 100) |
|  |
| Running the tests  *The two dots represent two passing tests.* | acc = Accountant() | # Test multiple deposits. |
| self.assertEqual(acc.balance, 0) |
| self.acc.deposit(100) |
| .. | # Test non-default balance. |  |
| self.acc.deposit(100) |
| --------------------------------------- | acc = Accountant(100) |
| self.assertEqual(self.acc.balance, 300) |
| Ran 2 tests in 0.000s | self.assertEqual(acc.balance, 100) |

|  |  |  |
| --- | --- | --- |
| OK | unittest.main() | def test\_withdrawal(self): |
| # Test single withdrawal. |
|  | Running the test | self.acc.deposit(1000) |
| self.acc.withdraw(100) |
|  | . |  |
| |  | | --- | | *Python provides a number of assert methods you can use to test your code.* | | self.assertEqual(self.acc.balance, 900) |
| --------------------------------------- | unittest.main() |
| Ran 1 test in 0.000s |
|  |  |  |
| |  | | --- | | Verify that a==b, or a != b | |  | Running the tests |
| assertEqual(a, b) | OK |  |
| assertNotEqual(a, b) |  | ... |
| --------------------------------------- |
| |  | | --- | | Verify that x is True, or x is False | |  | Ran 3 tests in 0.001s |
| *In general you shouldn’t modify a test once it’s written.* |
|  |
| assertTrue(x) | *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.* | OK |
| assertFalse(x) |
| |  | | --- | | Verify an item is in a list, or not in a list | | *If your original requirements have changed, it may be* |  |
|  |  |  |
| assertIn(item, list) | *appropriate to modify some tests. This usually happens in the early stages of a project when desired behavior is still being sorted out.* | |  | | --- | | *More cheat sheets available at* | |
| assertNotIn(item, list) |  |

|  |  |  |
| --- | --- | --- |
| |  | | --- | |  | | *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 |
| 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. | 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. |
| |  | | --- | |  |  |  | | --- | | *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/.*](http://ehmatthes.github.io/pcc/chapter_12/README.html) | | while True:   # Start event loop.  for event in pg.event.get(): if event.type == pg.QUIT: sys.exit()  # Refresh screen. |
| |  | | --- | | Pygame on Linux | | pg.display.flip() |
| $ sudo apt-get install python3-dev mercurial | run\_game() |
| libsdl-image1.2-dev libsdl2-dev libsdl-ttf2.0-dev  $ pip install --user | Setting a custom window size  *The display.set\_mode() function accepts a tuple that defines the screen size.* |
| hg+http://bitbucket.org/pygame/pygame | screen\_dim = (1200, 800) |
| |  | | --- | | Pygame on OS X | | screen = pg.display.set\_mode(screen\_dim) |
| *This assumes you’ve used Homebrew to install Python 3.* | Setting a custom background color |
| $ brew install hg sdl sdl\_image sdl\_ttf $ pip install --user | *Colors are defined as a tuple of red, green, and blue values. Each value ranges from 0-255.* |
| hg+http://bitbucket.org/pygame/pygame | bg\_color = (230, 230, 230) |
| |  | | --- | | Pygame on Windows | | screen.fill(bg\_color) |
| [*Find an installer at*](https://bitbucket.org/pygame/pygame/downloads/)  [*https://bitbucket.org/pygame/pygame/downloads/ or*](http://www.lfd.uci.edu/~gohlke/pythonlibs/#pygame) [*http://www.lfd.uci.edu/~gohlke/pythonlibs/#pygame*](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:* | |  | | --- | |  |  |  | | --- | | *Many objects in a game can be treated as simple*  *rectangles, rather than their actual shape. This simplifies* | |
| > python –m pip install --user   pygame-1.9.2a0-cp35-none-win32.whl | *code without noticeably affecting game play. Pygame has a rect object that makes it easy to work with game objects.* |
|  |  |
| |  | | --- | | 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* | | |  | | --- | | Getting the screen rect object  *We already have a screen object; we can easily access the rect object associated with the screen.* | |
| *successful.* | screen\_rect = screen.get\_rect() |
| $ python  >>> import pygame | |  | | --- | | 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* | *Pygame’s event loop registers an event any time the mouse moves, or a mouse button is pressed or released.* | |  | | --- | | Removing an item from a group | |
| *It’s important to delete elements that will never appear again in the* |
| *screen with the blit() method. The blit() method acts on the screen object, and takes the image object and image rect as arguments.* | |  | | --- | | Responding to the mouse button | | *game, so you don’t waste memory and resources.* |
|  |  | bullets.remove(bullet) |
|  | |  | | --- | | for event in pg.event.get(): | |  |
| # Draw ship to screen. | if event.type == pg.MOUSEBUTTONDOWN: | |  | | --- | |  | |
| screen.blit(ship, ship\_rect) |
| ship.fire\_bullet() |
|  |  |  |
| 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* | |  | | --- | | Finding the mouse position  *The mouse position is returned as a tuple.* | | |  | | --- | | *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.* | |
| *screen.* | |  | | --- | | mouse\_pos = pg.mouse.get\_pos() | |  |
|  |  |  |
| def blitme(self): | |  | | --- | | 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* | | |  | | --- | | 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.* | |
| """Draw ship at current location.""" |
| self.screen.blit(self.image, self.rect) |
|  |  |  |
|  |  | if pg.sprite.spritecollideany(ship, aliens): |
|  |  |  |
|  | |  | | --- | | if button\_rect.collidepoint(mouse\_pos): | | ships\_left -= 1 |
|  |  | |  | | --- | | Collisions between two groups | |
| |  | | --- | | *Pygame watches for events such as key presses and* | | start\_game() |
| *mouse actions. You can detect any event you care about in the event loop, and respond with any action that’s* | |  | | --- | | Hiding the mouse | | *The sprite.groupcollide() function takes two groups, and two booleans. The function returns a dictionary containing information* |
| *appropriate for your game.* | |  | | --- | | pg.mouse.set\_visible(False) | | *about the members that have collided. The booleans tell Pygame whether to delete the members of either group that have collided.* |
| |  | | --- | | Responding to key presses | | |  | | --- | |  | | collisions = pg.sprite.groupcollide( |
| *Pygame’s main event loop registers a KEYDOWN event any time a key is pressed. When this happens, you can check for specific keys.* | bullets, aliens, True, True) |
| *Pygame has a Group class which makes working with a group of similar objects easier. A group is like a list, with* |
| score += len(collisions) \* alien\_point\_value |
| for event in pg.event.get(): | *some extra functionality that’s helpful when building games.* |  |
| if event.type == pg.KEYDOWN: | |  | | --- | | Making and filling a group  *An object that will be placed in a group must inherit from Sprite.* | | |  | | --- | |  | |
| if event.key == pg.K\_RIGHT: |
| *You can use text for a variety of purposes in a game. For* |
| ship\_rect.x += 1 |
|  | from pygame.sprite import Sprite, Group | *example you can share information with players, and you can display a score.* |
| elif event.key == pg.K\_LEFT: |
| ship\_rect.x -= 1 |
| elif event.key == pg.K\_SPACE: | def Bullet(Sprite): | |  | | --- | | 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* | |
| ship.fire\_bullet() | ... |
| elif event.key == pg.K\_q: | def draw\_bullet(self): |
| sys.exit() | ... |
|  |
| |  | | --- | | Responding to released keys  *When the user releases a key, a KEYUP event is triggered.* | | def update(self): | *rect object associated with the image. We then center the image on the screen and display it.* |
| ... |
| if event.type == pg.KEYUP: | bullets = Group() | msg = "Play again?" |
| if event.key == pg.K\_RIGHT: | new\_bullet = Bullet() | msg\_color = (100, 100, 100) |
| ship.moving\_right = False | bg\_color = (230, 230, 230) |

|  |  |  |
| --- | --- | --- |
|  | bullets.add(new\_bullet) |  |
| |  | | --- | |  | | |  | | --- | | Looping through the items in a group  *The sprites() method returns all the members of a group.* | | f = pg.font.SysFont(None, 48) |
| msg\_image = f.render(msg, True, msg\_color, |
| *The Pygame documentation is really helpful when building* |
| bg\_color) |
| *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/.* | for bullet in bullets.sprites(): |  |
| msg\_image\_rect = msg\_image.get\_rect() |
| bullet.draw\_bullet() | msg\_image\_rect.center = screen\_rect.center |
|  |
| *The most useful part of the documentation are the pages about specific parts of Pygame, such as the Rect() class* | |  | | --- | | Calling update() on a group  *Calling update() on a group automatically calls update() on each* | | screen.blit(msg\_image, msg\_image\_rect) |
| *and the sprite module. You can find a list of these elements* | *member of the group.* | *More cheat sheets available at* |
| *at the top of the help pages.* | bullets.update() |  |
|  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | |  |  | | --- | --- | | |  | | --- | |  | | | |  | | --- | |  | |
|  | |  | | --- | | 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* | | |  | | --- | | 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.* | |
|  |
|  | |  | | --- | | *the size of each point.* | | import matplotlib.pyplot as plt |
|  | import matplotlib.pyplot as plt |  |
| x\_values = list(range(1000)) |
| x\_values = list(range(1000)) | squares = [x\*\*2 for x in x\_values] |
| squares = [x\*\*2 for x in x\_values] | plt.scatter(x\_values, squares, c=squares, |
| |  | | --- | |  | | plt.scatter(x\_values, squares, s=10) | cmap=plt.cm.Blues, edgecolor='none', |
| s=10) |
|  |
| Data visualization involves exploring data through | plt.show() |  |
| 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 | |  | | --- | |  | | plt.scatter(x\_values[0], squares[0], c='green', |
| edgecolor='none', s=100) |
| plt.scatter(x\_values[-1], squares[-1], c='red', |
| *Plots can be customized in a wide variety of ways. Just about any element of a plot can be customized.* |
| edgecolor='none', s=100) |
| a few simple visualizations. | |  | | --- | | Adding titles and labels, and scaling axes | | plt.title("Square Numbers", fontsize=24) |
|  |
|  | import matplotlib.pyplot as plt | *--snip--* |
| |  | | --- | |  | |  | |  | | --- | | Removing axes | |
| *matplotlib runs on all systems, but setup is slightly different depending on your OS. If the minimal instructions here* | x\_values = list(range(1000)) | *You can customize or remove axes entirely. Here’s how to access each axis, and hide it.* |
| squares = [x\*\*2 for x in x\_values] |
| *don’t work for you, see the more detailed instructions at* [*http://ehmatthes.github.io/pcc/.*](http://ehmatthes.github.io/pcc/) *You should also consider* [*installing the Anaconda distrub*](http://ehmatthes.github.io/pcc/)*ution of Python from* | plt.scatter(x\_values, squares, s=10) | plt.axes().get\_xaxis().set\_visible(False) |
|  | plt.axes().get\_yaxis().set\_visible(False) |
| [*https://continuum.io/downloads/,*](https://continuum.io/downloads/) *which includes matplotlib.* | plt.title("Square Numbers", fontsize=24) | |  | | --- | | Setting a custom figure size | |
| plt.xlabel("Value", fontsize=18) |
|  |
| matplotlib on Linux | plt.ylabel("Square of Value", fontsize=18) | *You can make your plot as big or small as you want. Before plotting your data, add the following code. The* dpi *argument is* |
| plt.tick\_params(axis='both', which='major', |
|  |
| |  | | --- | | $ sudo apt-get install python3-matplotlib | | labelsize=14) | *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)) |
| matplotlib on OS X  *Start a terminal session and enter* import matplotlib *to see if* |  |
|  | |  | | --- | | Saving a plot | |
| *it’s already installed on your system. If not, try this command:* | plt.show() |
|  |  |  |
| |  | | --- | | $ pip install --user matplotlib | | |  | | --- | | Using a colormap  *A colormap varies the point colors from one shade to another,* | | *The matplotlib viewer has an interactive save button, but you can also save your visualizations programmatically. To do so, replace* |
| matplotlib on Windows  *You first need to install Visual Studio, which you can do from* | *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* | plt.show() *with* plt.savefig()*. The* bbox\_inches='tight' *argument trims extra whitespace from the plot.* |
| [*https://dev.windows.com/.*](https://dev.windows.com/) [*The Communit*](https://pypi.python.org/pypi/matplotlib/)*y edition is free. Then go* [*to https://pypi.python.org/pypi/matplotlib/ or*](http://www.lfd.uic.edu/~gohlke/pythonlibs/#matplotlib) | *the* cmap *argument specifies which colormap to use.*  *The* edgecolor='none' *argument removes the black outline* | plt.savefig('squares.png', bbox\_inches='tight') |
| [*http://www.lfd.uic.edu/~gohlke/pythonlibs/#matplotlib*](http://www.lfd.uic.edu/~gohlke/pythonlibs/#matplotlib) *and download* | *from each point.* |  |
| *an appropriate installer file.* | plt.scatter(x\_values, squares, c=squares, |  |
| cmap=plt.cm.Blues, edgecolor='none', | *The matplotlib gallery and documentation are at* |
|  |  |  |
| |  | | --- | |  | | s=10) | [*http*](http://matplotlib.org/)[*://matpl*](http://matplotlib.org/api/pyplot_summary.html)[*otlib.org/.*](http://matplotlib.org/) *Be sure to visit the* [*examples,*](http://matplotlib.org/examples/) [*gallery,*](http://matplotlib.org/gallery.html)[*and*](http://matplotlib.org/) [*pyplot*](http://matplotlib.org/api/pyplot_summary.html) [*links.*](http://matplotlib.org/) |
|  |
| |  | | --- | | Making a line graph | |  |  |
| import matplotlib.pyplot as plt |  | |  |  |  |  | | --- | --- | --- | --- | | |  |  |  | | --- | --- | --- | | *Covers Python 3 and Python 2* | |  | | --- | |  | | | |
| x\_values = [0, 1, 2, 3, 4, 5] |
| squares = [0, 1, 4, 9, 16, 25] |
| plt.plot(x\_values, squares) |
| plt.show() |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | |  | | --- | |  | | |  |
|  |  |
| *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* | |  | | --- | | 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.* | | | *You can include as many individual graphs in one figure as you want. This is useful, for example, when comparing related datasets.* |
| *between two sets of data.* | Sharing an x-axis |
| Plotting two sets of data | *The following code plots a set of squares and a set of cubes on* |
| *Here we use* plt.scatter() *twice to plot square numbers and cubes on the same figure.* | %A | Weekday name, such as Monday | *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* |
| %B | Month name, such as January |
| import matplotlib.pyplot as plt | %m | Month, as a number (01 to 12) | *figure. The first two arguments control the number of rows and columns generated in the figure.* |
|  | %d | Day of the month, as a number (01 to 31) |
| %Y | Four-digit year, such as 2016 |
| x\_values = list(range(11)) | import matplotlib.pyplot as plt |
| squares = [x\*\*2 for x in x\_values] | %y | Two-digit year, such as 16 |
| %H | Hour, in 24-hour format (00 to 23) | x\_vals = list(range(11)) |
| cubes = [x\*\*3 for x in x\_values] |
| %I | Hour, in 12-hour format (01 to 12) |
| plt.scatter(x\_values, squares, c='blue', | squares = [x\*\*2 for x in x\_vals] |
| %p | AM or PM |
| cubes = [x\*\*3 for x in x\_vals] |
| edgecolor='none', s=20) | %M | Minutes (00 to 59) |
| %S | Seconds (00 to 61) | fig, axarr = plt.subplots(2, 1, sharex=True) |
| plt.scatter(x\_values, cubes, c='red', |
| edgecolor='none', s=20) | |  | | --- | | Converting a string to a datetime object | | |  |
|  |
| plt.axis([0, 11, 0, 1100]) | new\_years = dt.strptime('1/1/2017', '%m/%d/%Y') | | axarr[0].scatter(x\_vals, squares) |
| axarr[0].set\_title('Squares') |
|  |
| plt.show() | |  | | --- | | Converting a datetime object to a string | | |  |
|  |  |
| 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* | ny\_string = dt.strftime(new\_years, '%B %d, %Y') | | axarr[1].scatter(x\_vals, cubes, c='red') |
| axarr[1].set\_title('Cubes') |
| print(ny\_string) | |
| *takes a* facecolor *to use for the fill, and an optional* alpha *argument that controls the color’s transparency.* | |  | | --- | | 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.* | | | plt.show() |
|  |  |
| plt.fill\_between(x\_values, cubes, squares, | Sharing a y-axis  *To share a y-axis, we use the* sharey=True *argument.* |
| facecolor='blue', alpha=0.25) |
|  |  |
|  | from datetime import datetime as dt | | import matplotlib.pyplot as plt |
| |  | | --- | |  | | x\_vals = list(range(11)) |
| *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.* | import matplotlib.pyplot as plt | |
| squares = [x\*\*2 for x in x\_vals] |
| from matplotlib import dates as mdates | |
| cubes = [x\*\*3 for x in x\_vals] |
| dates = [ | |
| |  | | --- | | Generating the current date  *The* datetime.now() *function returns a datetime object representing the current date and time.* | | fig, axarr = plt.subplots(1, 2, sharey=True) |
| dt(2016, 6, 21), dt(2016, 6, 22), | |
| dt(2016, 6, 23), dt(2016, 6, 24), | | axarr[0].scatter(x\_vals, squares) |
| ] | |
|  |
| from datetime import datetime as dt | highs = [57, 68, 64, 59] | | axarr[0].set\_title('Squares') |
| axarr[1].scatter(x\_vals, cubes, c='red') |
| today = dt.now() | fig = plt.figure(dpi=128, figsize=(10,6)) | | axarr[1].set\_title('Cubes') |
| date\_string = dt.strftime(today, '%m/%d/%Y') |
| plt.plot(dates, highs, c='red') | | plt.show() |
| print(date\_string) |
| plt.title("Daily High Temps", fontsize=24) | |
| |  | | --- | | 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.* | |  |
| plt.ylabel("Temp (F)", fontsize=16) | |
| x\_axis = plt.axes().get\_xaxis() | |
| x\_axis.set\_major\_formatter( | |
| mdates.DateFormatter('%B %d %Y') | |
| from datetime import datetime as dt |  |
| ) | |
|  | fig.autofmt\_xdate() | |  |
| new\_years = dt(2017, 1, 1) | |  | | --- | | *More cheat sheets available at* | |
| fall\_equinox = dt(year=2016, month=9, day=22) | plt.show() | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | |  | | --- | |  | | |  | | --- | |  | |
|  |
|  | |  | | --- | | Making a scatter plot  *The data for a scatter plot needs to be a list containing tuples of* | | *You can add as much data as you want when making a visualization.* |
|  | |  | | --- | | Plotting squares and cubes | |
| |  | | --- | |  | | *the form (x, y). The* stroke=False *argument tells Pygal to make an XY chart with no line connecting the points.* |
|  |  |  |
|  | import pygal | import pygal |
| |  |  | | --- | --- | | |  | | --- | |  | | | squares = [ | x\_values = list(range(11)) |
| squares = [x\*\*2 for x in x\_values] |
|  |
| Data visualization involves exploring data through visual representations. Pygal helps you make visually | (0, 0), (1, 1), (2, 4), (3, 9), | cubes = [x\*\*3 for x in x\_values] |
| (4, 16), (5, 25), |
| ] | chart = pygal.Line() |
| 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. |
| chart = pygal.XY(stroke=False) | chart.force\_uri\_protocol = 'http' |
| chart.title = "Squares and Cubes" |
| chart.force\_uri\_protocol = 'http' |
| chart.x\_labels = x\_values |
| chart.add('x^2', squares) |
|  |
|  | chart.render\_to\_file('squares.svg') | chart.add('Squares', squares) |
| |  | | --- | |  | | |  | | --- | | Using a list comprehension for a scatter plot  *A list comprehension can be used to effficiently make a dataset for* | | chart.add('Cubes', cubes) |
| *Pygal can be installed using pip.* | chart.render\_to\_file('squares\_cubes.svg') |
|  |
|  | *a scatter plot.* |
|  |
| |  | | --- | | Pygal on Linux and OS X | | squares = [(x, x\*\*2) for x in range(1000)] | |  | | --- | | *Pygal allows you to fill the area under or over each series of data.* | |
|  |  |  |
| |  | | --- | | $ pip install --user pygal | | |  | | --- | | Making a bar graph | | *The default is to fill from the x-axis up, but you can fill from any horizontal line using the argument.* |
| |  | | --- | | Pygal on Windows | | *A bar graph requires a list of values for the bar sizes. To label the* | zero |
|  | *bars, pass a list of the same length to .* | chart = pygal.Line(fill=True, zero=0) |
| |  | | --- | | > python –m pip install --user pygal | | x\_labels |  |
|  |  |  |
|  |  |  |
| |  | | --- | |  | | outcomes = [1, 2, 3, 4, 5, 6] |  |
| *To make a plot with Pygal, you specify the kind of plot and then add the data.* |
| frequencies = [18, 16, 18, 17, 18, 13] |
| |  | | --- | | Making a line graph  *To view the output, open the file squares.svg in a browser.* | | chart = pygal.Bar() |  |
| chart.force\_uri\_protocol = 'http' |
| chart.x\_labels = outcomes |
| import pygal |  |  |
| chart.add('D6', frequencies) |
|  | chart.render\_to\_file('rolling\_dice.svg') |  |
| x\_values = [0, 1, 2, 3, 4, 5] | |  | | --- | | 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* | |  |
| squares = [0, 1, 4, 9, 16, 25] | |  | | --- | | [*The documentation fo*](http://www.pygal.org/)*r Pygal is available at* [*http://www.pygal.org/.*](http://www.pygal.org/) | |
| chart = pygal.Line() |
|  |  | |  | | --- | |  | |
| chart.force\_uri\_protocol = 'http' | *height of each bar.* |
| cart.d('x^2', suar) |
| cart.d('x^2', suar) | import pygal |  |
| *If you’re viewing svg output in a browser, Pygal needs to* |
| chart.render\_to\_file('squares.svg') |
| |  | | --- | | Adding labels and a title | | results = { | *render the output file in a specific way. The*  force\_uri\_protocol *attribute for chart objects needs to* |
| *--snip--* | 1:18, 2:16, 3:18, | *be set to* 'http'. |
| :1 518,:1 |
| chart = pygal.Line() |  | |  |  |  |  | | --- | --- | --- | --- | | |  |  |  | | --- | --- | --- | | *Covers Python 3 and Python 2* | |  | | --- | |  | | | |
| } |
| chart.force\_uri\_protocol = 'http' |
| chart.title = "Squares" | chart = pygal.Bar() |
| chart.x\_labels = x\_values |
| chart.force\_uri\_protocol = 'http' |
| chart.x\_title = "Value" |
| chart.x\_labels = results.keys() |
| chart.y\_title = "Square of Value" |
| chart.add('D6', results.values()) |
| chart.add('x^2', squares) |
| chart.render\_to\_file('rolling\_dice.svg') |
| chart.render\_to\_file('squares.svg') |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |  |
|  |  |
| *Pygal lets you customize many elements of a plot. There are some excellent default themes, and many options for styling individual plot elements.* | |  | | --- | | Configuration settings  *Some settings are controlled by a* Config *object.* | | *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* |
| |  | | --- | | Using built-in styles | | my\_config = pygal.Config() | *over each country on the map.* |
| *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.* | my\_config.show\_y\_guides = False | Installing the world map module  *The world map module is not included by default in Pygal 2.0. It can be installed with pip:* |
| my\_config.width = 1000 |
| my\_config.dots\_size = 5 |
| import pygal | chart = pygal.Line(config=my\_config) | $ pip install --user pygal\_maps\_world |
|  |
| from pygal.style import LightGreenStyle | *--snip--* | Making a world map |
| x\_values = list(range(11)) | |  | | --- | | Styling series  *You can give each series on a chart different style settings.* | | *The following code makes a simple world map showing the countries of North America.* |
| squares = [x\*\*2 for x in x\_values] |
| cubes = [x\*\*3 for x in x\_values] | chart.add('Squares', squares, dots\_size=2) | from pygal.maps.world import World |
| chart\_style = LightGreenStyle() | chart.add('Cubes', cubes, dots\_size=3) | wm = World() |
|  | |  | | --- | | 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.* | |  |
| chart = pygal.Line(style=chart\_style) | wm.force\_uri\_protocol = 'http' |
| chart.force\_uri\_protocol = 'http' | wm.title = 'North America' |
| chart.title = "Squares and Cubes" | wm.add('North America', ['ca', 'mx', 'us']) |
| chart.x\_labels = x\_values |
|  | import pygal | wm.render\_to\_file('north\_america.svg') |
| chart.add('Squares', squares) | repos = [ | 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* |
| chart.add('Cubes', cubes) |
| chart.render\_to\_file('squares\_cubes.svg') | { |
| |  | | --- | | Parametric built-in styles | | 'value': 20506, | *and its code.* |
|  |
| *Some built-in styles accept a custom color, then generate a theme based on that color.* | 'color': '#3333CC', | from pygal.maps.world import COUNTRIES |
| 'xlink': 'http://djangoproject.com/', |
|  |
| from pygal.style import LightenStyle | }, | for code in sorted(COUNTRIES.keys()): |
| 20054, |
| print(code, COUNTRIES[code]) |
| 12607, |
| *--snip--* | 11827, | Plotting numerical data on a world map  *To plot numerical data on a map, pass a dictionary to* add() *instead of a list.* |
| chart\_style = LightenStyle('#336688') | ] |
| chart = pygal.Line(style=chart\_style) |
|  |
| *--snip--* |
|  | chart = pygal.Bar() |  |
| |  | | --- | | Customizing individual style properties  *Style objects have a number of properties you can set individually.* | | chart.force\_uri\_protocol = 'http' | from pygal.maps.world import World |
| chart.x\_labels = [ | populations = { |
| 'django', 'requests', 'scikit-learn', |
| chart\_style = LightenStyle('#336688') |  | 'ca': 34126000, |
| 'tornado', |
| 'us': 309349000, |
| chart\_style.plot\_background = '#CCCCCC' | ] |
| chart\_style.major\_label\_font\_size = 20 | 'mx': 113423000, |
| chart.y\_title = 'Stars' |
| chart\_style.label\_font\_size = 16 | } |
| chart.add('Python Repos', repos) |
| *--snip--* | chart.render\_to\_file('python\_repos.svg') |  |
| |  | | --- | | Custom style class  *You can start with a bare style class, and then set only the properties you care about.* | |  | wm = World() |
| wm.force\_uri\_protocol = 'http' |
| wm.title = 'Population of North America' |
| wm.add('North America', populations) |
| chart\_style = Style() |  | wm.render\_to\_file('na\_populations.svg') |
| chart\_style.colors = [ |
| '#CCCCCC', '#AAAAAA', '#888888'] |
| chart\_style.plot\_background = '#EEEEEE' |
| chart = pygal.Line(style=chart\_style) |  | |  | | --- | | *More cheat sheets available at* | |
| *--snip--* |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | |  | | --- | |  |  |  | | --- | | *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 |
| 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. | 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) |
| |  | | --- | |  |  |  | | --- | | *It’s usualy best to install Django to a virtual environment,* | | def \_\_str\_\_(self):   return self.text |
| *where your project can be isolated from your other Python projects. Most commands assume you’re working in an active virtual environment.* | |  | | --- | | 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.* | |
| |  | | --- | | Create a virtual environment | | INSTALLED\_APPS = ( |
| |  | | --- | | $ python –m venv ll\_env | | *--snip--*   'django.contrib.staticfiles', |
| |  | | --- | | Activate the environment (Linux and OS X) | | # My apps |
| |  | | --- | | $ source ll\_env/bin/activate | | 'learning\_logs', |
| |  | | --- | | Activate the environment (Windows) | | ) |
|  |  |
| |  | | --- | | > ll\_env\Scripts\activate | | |  | | --- | | Migrating the database  *The database needs to be modified to store the kind of data that* | |
| |  | | --- | | Install Django to the active environment | | *the model represents.* |
| |  | | --- | | (ll\_env)$ pip install Django | | $ python manage.py makemigrations learning\_logs $ python manage.py migrate |
|  |  |
| |  | | --- | |  |  |  | | --- | | *To start a project we’ll create a new project, create a database, and start a development server.* | | |  | | --- | | Creating a superuser  *A superuser is a user account that has access to all aspects of the project.* | |
|  |  |
| |  | | --- | | Create a new project | | $ python manage.py createsuperuser |
|  | |  | | --- | | Registering a model | |
| $ django-admin.py startproject learning\_log . | *You can register your models with Django’s admin site, which* |
| |  | | --- | | Create a database | | *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* |
| $ python manage.py migrate | *http://localhost:8000/admin/.* |
| |  | | --- | | View the project  *After issuing this command, you can view the project at http://localhost:8000/.* | | from django.contrib import admin  from learning\_logs.models import Topic |
| $ python manage.py runserver | admin.site.register(Topic) |

|  |  |
| --- | --- |
| |  | | --- | | Create a new app  *A Django project is made up of one or more apps.* | |

|  |
| --- |
| $ python manage.py startapp learning\_logs |

|  |
| --- |
|  |

|  |
| --- |
| *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*](http://docs.djangoproject.com/) *is available at*  [*http://docs.djangoproject.com/.*](http://docs.djangoproject.com/) *The Django documentation* [*is thorough and user-friendly, s*](http://docs.djangoproject.com/)*o check it out!* |

|  |  |  |  |
| --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | *Covers Python 3 and Python 2* | |  | | --- | |  | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |  |
|  |
| |  | | --- | | Writing a simple template | | |  | | --- | | *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.* | | 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.* |
| *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.* | |  |  | | --- | --- | | |  | | --- | | Defining a model with a foreign key | | | *The vertical line after a template variable indicates a filter. In this case a filter called* date *formats date objects, and the filter* |
| <p>Learning Log</p> | class Entry(models.Model): | linebreaks *renders paragraphs properly on a web page.* |
|  | """Learning log entries for a topic.""" | {% extends 'learning\_logs/base.html' %} |
| <p>Learning Log helps you keep track of your | topic = models.ForeignKey(Topic) |
| text = models.TextField() | {% block content %} |
| learning, for any topic you're learning |
| date\_added = models.DateTimeField( |
| about.</p> |  |
| auto\_now\_add=True) |
|  |  | <p>Topic: {{ topic }}</p> |
| |  | | --- | |  | | def \_\_str\_\_(self): | <p>Entries:</p> |
| *Many elements of a web page are repeated on every page* | return self.text[:50] + "..." |
| <ul> |
| *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.* | |  | | --- | |  | | {% for entry in entries %} |
| <li> |
| *Most pages in a project need to present data that’s specific* | <p> |
| |  | | --- | | The parent template | | *to the current user.* | {{ entry.date\_added|date:'M d, Y H:i' }} |
| </p> |
| *The parent template defines the elements common to a set of pages, and defines blocks that will be filled by individual pages.* | |  | | --- | | URL parameters  *A URL often needs to accept a parameter telling it which data to* | |  |
| <p> |
|  |
| <p> | *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*.* | {{ entry.text|linebreaks }} |
| </p> |
| <a href="{% url 'learning\_logs:index' %}"> | </li> |
| Learning Log |
| {% empty %} |
| </a> | urlpatterns = [ | <li>There are no entries yet.</li> |
| </p> |
| url(r'^$', views.index, name='index'), | {% endfor %} |
| {% block content %}{% endblock content %} | url(r'^topics/(?P<topic\_id>\d+)/$', | </ul> |
| views.topic, name='topic'), |
| |  | | --- | | The child template | | ] | {% endblock content %} |
| {% extends %} |  |  |
| *the structure of the parent template. It then defines the content for any blocks defined in the parent template.* | |  | | --- | | 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 | |  |
| {% extends 'learning\_logs/base.html' %} | *dictionary to the template, containing data that should be displayed on the page.* | *You can explore the data in your project from the command line. This is helpful for developing queries and testing code* |
| {% block content %} | def topic(request, topic\_id): | *snippets.* |
| <p> | """Show a topic and all its entries.""" | Start a shell session |
| Learning Log helps you keep track | toic Tocs.bjtsgetd=picid |
|  |  | $ python manage.py shell |
| of your learning, for any topic you're | entries = topic.entry\_set.order\_by( |
|  |
| learning about. | '-date\_added') | Access data from the project |
| </p> | context = { |
| {% endblock content %} |  | >>> from learning\_logs.models import Topic |
| 'topic': topic, |
| 'entries': entries, | >>> Topic.objects.all() |
| } | [<Topic: Chess>, <Topic: Rock Climbing>] |
| return render(request, | >>> topic = Topic.objects.get(id=1) |
| 'learning\_logs/topic.html', context) | >>> topic.text |
|  |  | 'Chess' |
|  | |  | | --- | |  | |  |
| *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.* | *If you make a change to your project and the change doesn’t seem to have any effect, try restarting the server:* | |  | | --- | | *More cheat sheets available at* | |
|  |
| $ python manage.py runserver |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | |  | | --- | |  | |
|  | |  | | --- | | 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.* | | |  | | --- | | 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,* | |
|  |
|  | from django.conf.urls import url | *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* |
|  |
| from django.contrib.auth.views import login |
| from . import views | *who have logged in. Users who haven’t logged in see links to register or log in.* |
|  |
|  | urlpatterns = [ | <p> |
| url(r'^login/$', login, | <a href="{% url 'learning\_logs:index' %}"> |
| Most web applications need to let users create accounts. This lets users create and work with their own data. Some of this data may be private, and |  |  |
| {'template\_name': 'users/login.html'}, |
| Learning Log |
| name='login'), | </a> |
| url(r'^logout/$', views.logout\_view, |
| {% if user.is\_authenticated %} |
| some may be public. Django’s forms allow users to enter and modify their data. | name='logout'), | Hello, {{ user.username }}. |
| url(r'^register/$', views.register, |
| <a href="{% url 'users:logout' %}"> |
| name='register'), | log out |
|  | ] |  |
| </a> |
|  | |  | | --- | | 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* | | {% else %} |
| |  | | --- | | *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.* | |
| <a href="{% url 'users:register' %}"> |
| register |
| </a> - |
| |  | | --- | | Making a users app  *After making the app, be sure to add* 'users' *to* INSTALLED\_APPS *in the project’s settings.py file.* | | *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* | <a href="{% url 'users:login' %}"> |
| log in |
| </a> |
|  |
| $ python manage.py startapp users | *default login form in paragraph format. The* <input> *element named redirects the user to the home page after a successful* | {% endif %} |
| </p> |
| |  | | --- | | Including URLS for the users app | | |  |  | | --- | --- | | *login.* | next | |  |
| *Add a line to the project’s urls.py file so the* users *app’s URLs are* | { eend "aringogbas.hl"%} | {% block content %}{% endblock content %} |
| *included in the project.* | { eend "aringogbas.hl"%} | |  | | --- | | The logout view | |
| urlpatterns = [ | {% block content %} | *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.* |
| url(r'^admin/', include(admin.site.urls)), | {% if form.errors %} |
| url(r'^users/', include('users.urls', | <p> |
| namespace='users')), | Your username and password didn't match. |
| url(r'', include('learning\_logs.urls', |  |
| Please try again. | |  | | --- | | from django.http import HttpResponseRedirect | |
| namespace='learning\_logs')), | </p> | from django.core.urlresolvers import reverse |
| ] | {% endif %} | from django.contrib.auth import logout |
|  |  |
|  | <form method="post" | def logout\_view(request): |
|  | action="{% url 'users:login' %}"> | """Log the user out.""" |
| {% csrf token %} | logout(request) |
| *There are a number of ways to create forms and work with* |
| {{ form.as\_p }} | return HttpResponseRedirect( |
| *them. You can use Django’s defaults, or completely customize your forms. For a simple way to let users enter* | <button name="submit">log in</button> | reverse('learning\_logs:index')) |
| *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* | <input type="hidden" name="next" | |  |  |  |  | | --- | --- | --- | --- | | |  |  |  | | --- | --- | --- | | *Covers Python 3 and Python 2* | |  | | --- | |  | | | |
| value="{% url 'learning\_logs:index' %}"/> |
| </form> |
| *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.* | {% endblock content %} |

|  |  |  |  |
| --- | --- | --- | --- |
|  | |  | | --- | |  | |  |
| The register view  *The register view needs to display a blank registration form when the page is first requested, and then process completed* | |  | | --- | | The register template  *The register template displays the registration form in paragraph formats.* | | 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.* |
| *registration forms. A successful registration logs the user in and redirects to the home page.* | |  | | --- | | {% extends 'learning\_logs/base.html' %} | | *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 import login | {% block content %} | from django.contrib.auth.decorators import / |
| from django.contrib.auth import authenticate | login\_required |
| <form method='post' |
| from django.contrib.auth.forms import \ | *--snip--* |
| UserCreationForm | @login\_required |
| action="{% url 'users:register' %}"> |
| def register(request): | {% csrf\_token %} | def topic(request, topic\_id): |
| """Register a new user.""" | {{ form.as\_p }} | """Show a topic and all its entries.""" |
|  |
| if request.method != 'POST': | <button name='submit'>register</button> | Setting the redirect URL  *The* @login\_required *decorator sends unauthorized users to the* |
| # Show blank registration form. |
| form = UserCreationForm() | <input type='hidden' name='next' |
| *login page. Add the following line to your project’s settings.py file so Django will know how to find your login page.* |
| else: | value="{% url 'learning\_logs:index' %}"/> |
| # Process completed form. |
|  | </form> | LOGIN\_URL = '/users/login/' |
| form = UserCreationForm( |
| data=request.POST) | {% endblock content %} | Preventing inadvertent access  *Some pages serve data based on a parameter in the URL. You* |
| if form.is\_valid(): |
|  |  |  |
| new\_user = form.save() | |  | | --- | |  | | *return a 404 error if they don’t. Here’s an example view.* |
|  |
| # Log in, redirect to home page. |  | from django.http import Http404 |
| pw = request.POST['password1'] | *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.* |
| authenticated\_user = authenticate( | *--snip--* |
| username=new\_user.username, |
|  |
| password=pw | |  | | --- | | 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.* | | def topic(request, topic\_id): |
| """Show a topic and all its entries.""" |
| ) |
| topic = Topics.objects.get(id=topic\_id) |
| login(request, authenticated\_user) |
| return HttpResponseRedirect( | if topic.owner != request.user: |
| raise Http404 |
| reverse('learning\_logs:index')) |
| *--snip--* |
|  |
|  |  |  |
|  |  |  |
| return render(request, | from django.db import models |  |
| 'users/register.html', context) | from django.contrib.auth.models import User |
|  |
|  | class Topic(models.Model): |  |
| *with the user’s existing data. Users can then modify and save their data.* |
| """A topic the user is learning about.""" |
| *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* | text = models.CharField(max\_length=200) | Creating a form with initial data  *The instance parameter allows you to specify initial data for a form.* |
| date\_added = models.DateTimeField( |
| auto\_now\_add=True) |
| [*style individual elements on a page. Lea*](http://django-bootstrap3.readthedocs.io/)*rn more at* [*http://django-bootstrap3.readthedocs.io/.*](http://django-bootstrap3.readthedocs.io/) |  | form = EntryForm(instance=entry) |
| owner = models.ForeignKey(User) |
|  |  | Modifying data before saving |
|  | def \_\_str\_\_(self): | *The argument* commit=False *allows you to make changes before writing data to the database.* |
| return self.text |
| *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* | |  | | --- | | 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.* | | new\_topic = form.save(commit=False) |
| new\_topic.owner = request.user |
| new\_topic.save() |
| *deployment process without any commitment. You’ll need* |
| *to install a set of he*[*roku tools, and use git to track*](http://devcenter.heroku.com/) *the state of your project. See* [*http://devcenter.heroku.com/,*](http://devcenter.heroku.com/) *and click on the Python link.* | topics = Topic.objects.filter( | |  | | --- | | *More cheat sheets available at* | |
| owner=request.user) |  |