

Beginner's Python Cheat Sheet

Variables and Strings

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

Hello world

```
print("Hello world!")
```

Hello world with a variable

```
msg = "Hello world!"  
print(msg)
```

Concatenation (combining strings)

```
first_name = 'albert'  
last_name = 'einstein'  
full_name = first_name + ' ' + last_name  
print(full_name)
```

Lists

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

Make a list

```
bikes = ['trek', 'redline', 'giant']
```

Get the first item in a list

```
first_bike = bikes[0]
```

Get the last item in a list

```
last_bike = bikes[-1]
```

Looping through a list

```
for bike in bikes:  
    print(bike)
```

Adding items to a list

```
bikes = []  
bikes.append('trek')  
bikes.append('redline')  
bikes.append('giant')
```

Making numerical lists

```
squares = []  
for x in range(1, 11):  
    squares.append(x**2)
```

Lists (cont.)

List comprehensions

```
squares = [x**2 for x in range(1, 11)]
```

Slicing a list

```
finishers = ['sam', 'bob', 'ada', 'bea']  
first_two = finishers[:2]
```

Copying a list

```
copy_of_bikes = bikes[:]
```

Tuples

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

Making a tuple

```
dimensions = (1920, 1080)
```

If statements

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

Conditional tests

equals	x == 42
not equal	x != 42
greater than	x > 42
or equal to	x >= 42
less than	x < 42
or equal to	x <= 42

Conditional test with lists

```
'trek' in bikes  
'surly' not in bikes
```

Assigning boolean values

```
game_active = True  
can_edit = False
```

A simple if test

```
if age >= 18:  
    print("You can vote!")
```

If-elif-else statements

```
if age < 4:  
    ticket_price = 0  
elif age < 18:  
    ticket_price = 10  
else:  
    ticket_price = 15
```

Dictionaries

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

A simple dictionary

```
alien = {'color': 'green', 'points': 5}
```

Accessing a value

```
print("The alien's color is " + alien['color'])
```

Adding a new key-value pair

```
alien['x_position'] = 0
```

Looping through all key-value pairs

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

Looping through all keys

```
fav_numbers = {'eric': 17, 'ever': 4}  
for name in fav_numbers.keys():  
    print(name + ' loves a number')
```

Looping through all the values

```
fav_numbers = {'eric': 17, 'ever': 4}  
for number in fav_numbers.values():  
    print(str(number) + ' is a favorite')
```

User input

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

Prompting for a value

```
name = input("What's your name? ")  
print("Hello, " + name + "!")
```

Prompting for numerical input

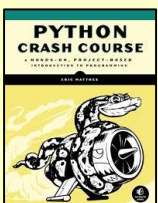
```
age = input("How old are you? ")  
age = int(age)
```

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

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



While loops

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

A simple while loop

```
current_value = 1
while current_value <= 5:
    print(current_value)
    current_value += 1
```

Letting the user choose when to quit

```
msg = ''
while msg != 'quit':
    msg = input("What's your message? ")
    print(msg)
```

Functions

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

A simple function

```
def greet_user():
    """Display a simple greeting."""
    print("Hello!")
```

```
greet_user()
```

Passing an argument

```
def greet_user(username):
    """Display a personalized greeting."""
    print("Hello, " + username + "!")
```

```
greet_user('jesse')
```

Default values for parameters

```
def make_pizza(topping='bacon'):
    """Make a single-topping pizza."""
    print("Have a " + topping + " pizza!")
```

```
make_pizza()
make_pizza('pepperoni')
```

Returning a value

```
def add_numbers(x, y):
    """Add two numbers and return the sum."""
    return x + y
```

```
sum = add_numbers(3, 5)
print(sum)
```

Classes

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

Creating a dog class

```
class Dog():
    """Represent a dog."""

    def __init__(self, name):
        """Initialize dog object."""
        self.name = name

    def sit(self):
        """Simulate sitting."""
        print(self.name + " is sitting.")
```

```
my_dog = Dog('Peso')
```

```
print(my_dog.name + " is a great dog!")
my_dog.sit()
```

Inheritance

```
class SARDog(Dog):
    """Represent a search dog."""

    def __init__(self, name):
        """Initialize the sardog."""
        super().__init__(name)

    def search(self):
        """Simulate searching."""
        print(self.name + " is searching.")
```

```
my_dog = SARDog('Willie')
```

```
print(my_dog.name + " is a search dog.")
my_dog.sit()
my_dog.search()
```

Infinite Skills

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

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

Working with files

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

Reading a file and storing its lines

```
filename = 'siddhartha.txt'
with open(filename) as file_object:
    lines = file_object.readlines()

for line in lines:
    print(line)
```

Writing to a file

```
filename = 'journal.txt'
with open(filename, 'w') as file_object:
    file_object.write("I love programming.")
```

Appending to a file

```
filename = 'journal.txt'
with open(filename, 'a') as file_object:
    file_object.write("\nI love making games.")
```

Exceptions

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

Catching an exception

```
prompt = "How many tickets do you need? "
num_tickets = input(prompt)

try:
    num_tickets = int(num_tickets)
except ValueError:
    print("Please try again.")
else:
    print("Your tickets are printing.")
```

Zen of Python

Simple is better than complex

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

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet - Lists

What are lists?

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

Defining a list

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

Making a list

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

Accessing elements

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

Getting the first element

```
first_user = users[0]
```

Getting the second element

```
second_user = users[1]
```

Getting the last element

```
newest_user = users[-1]
```

Modifying individual items

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

Changing an element

```
users[0] = 'valerie'  
users[-2] = 'ronald'
```

Adding elements

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

Adding an element to the end of the list

```
users.append('amy')
```

Starting with an empty list

```
users = []  
users.append('val')  
users.append('bob')  
users.append('mia')
```

Inserting elements at a particular position

```
users.insert(0, 'joe')  
users.insert(3, 'bea')
```

Removing elements

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

Deleting an element by its position

```
del users[-1]
```

Removing an item by its value

```
users.remove('mia')
```

Popping elements

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

Pop the last item from a list

```
most_recent_user = users.pop()  
print(most_recent_user)
```

Pop the first item in a list

```
first_user = users.pop(0)  
print(first_user)
```

List length

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

Find the length of a list

```
num_users = len(users)  
print("We have " + str(num_users) + " users.")
```

Sorting a list

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

Sorting a list permanently

```
users.sort()
```

Sorting a list permanently in reverse alphabetical order

```
users.sort(reverse=True)
```

Sorting a list temporarily

```
print(sorted(users))  
print(sorted(users, reverse=True))
```

Reversing the order of a list

```
users.reverse()
```

Looping through a list

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

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

Printing all items in a list

```
for user in users:  
    print(user)
```

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

```
for user in users:  
    print("Welcome, " + user + "!")  
  
print("Welcome, we're glad to see you all!")
```

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



The range() function

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

Printing the numbers 0 to 1000

```
for number in range(1001):  
    print(number)
```

Printing the numbers 1 to 1000

```
for number in range(1, 1001):  
    print(number)
```

Making a list of numbers from 1 to a million

```
numbers = list(range(1, 1000001))
```

Simple statistics

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

Finding the minimum value in a list

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

Finding the maximum value

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

Finding the sum of all values

```
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]  
total_years = sum(ages)
```

Slicing a list

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

Getting the first three items

```
finishers = ['kai', 'abe', 'ada', 'gus', 'zoe']  
first_three = finishers[:3]
```

Getting the middle three items

```
middle_three = finishers[1:4]
```

Getting the last three items

```
last_three = finishers[-3:]
```

Copying a list

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

Making a copy of a list

```
finishers = ['kai', 'abe', 'ada', 'gus', 'zoe']  
copy_of_finishers = finishers[:]
```

List comprehensions

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

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

Using a loop to generate a list of square numbers

```
squares = []  
for x in range(1, 11):  
    square = x**2  
    squares.append(square)
```

Using a comprehension to generate a list of square numbers

```
squares = [x**2 for x in range(1, 11)]
```

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

```
names = ['kai', 'abe', 'ada', 'gus', 'zoe']  
  
upper_names = []  
for name in names:  
    upper_names.append(name.upper())
```

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

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

Styling your code

Readability counts

- Use four spaces per indentation level.
- Keep your lines to 79 characters or fewer.
- Use single blank lines to group parts of your program visually.

Tuples

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

Defining a tuple

```
dimensions = (800, 600)
```

Looping through a tuple

```
for dimension in dimensions:  
    print(dimension)
```

Overwriting a tuple

```
dimensions = (800, 600)  
print(dimensions)
```

```
dimensions = (1200, 900)
```

Visualizing your code

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

Build a list and print the items in the list

```
dogs = []  
dogs.append('willie')  
dogs.append('hootz')  
dogs.append('peso')  
dogs.append('goblin')
```

```
for dog in dogs:  
    print("Hello " + dog + "!")  
print("I love these dogs!")
```

```
print("\nThese were my first two dogs:")  
old_dogs = dogs[:2]  
for old_dog in old_dogs:  
    print(old_dog)
```

```
del dogs[0]  
dogs.remove('peso')  
print(dogs)
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Dictionaries

What are dictionaries?

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

Defining a dictionary

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

Making a dictionary

```
alien_0 = {'color': 'green', 'points': 5}
```

Accessing values

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

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

Getting the value associated with a key

```
alien_0 = {'color': 'green', 'points': 5}
```

```
print(alien_0['color'])  
print(alien_0['points'])
```

Getting the value with `get()`

```
alien_0 = {'color': 'green'}
```

```
alien_color = alien_0.get('color')  
alien_points = alien_0.get('points', 0)
```

```
print(alien_color)  
print(alien_points)
```

Adding new key-value pairs

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

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

Adding a key-value pair

```
alien_0 = {'color': 'green', 'points': 5}
```

```
alien_0['x'] = 0  
alien_0['y'] = 25  
alien_0['speed'] = 1.5
```

Adding to an empty dictionary

```
alien_0 = {}  
alien_0['color'] = 'green'  
alien_0['points'] = 5
```

Modifying values

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

Modifying values in a dictionary

```
alien_0 = {'color': 'green', 'points': 5}  
print(alien_0)
```

```
# Change the alien's color and point value.  
alien_0['color'] = 'yellow'  
alien_0['points'] = 10  
print(alien_0)
```

Removing key-value pairs

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

Deleting a key-value pair

```
alien_0 = {'color': 'green', 'points': 5}  
print(alien_0)
```

```
del alien_0['points']  
print(alien_0)
```

Visualizing dictionaries

Try running some of these examples on pythontutor.com.

Looping through a dictionary

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

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

Looping through all key-value pairs

```
# Store people's favorite languages.  
fav_languages = {  
    'jen': 'python',  
    'sarah': 'c',  
    'edward': 'ruby',  
    'phil': 'python',  
}
```

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

Looping through all the keys

```
# Show everyone who's taken the survey.  
for name in fav_languages.keys():  
    print(name)
```

Looping through all the values

```
# Show all the languages that have been chosen.  
for language in fav_languages.values():  
    print(language)
```

Looping through all the keys in order

```
# Show each person's favorite language,  
# in order by the person's name.  
for name in sorted(fav_languages.keys()):  
    print(name + ": " + language)
```

Dictionary length

You can find the number of key-value pairs in a dictionary.

Finding a dictionary's length

```
num_responses = len(fav_languages)
```

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



Nesting — A list of dictionaries

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

Storing dictionaries in a list

```
# Start with an empty list.
users = []

# Make a new user, and add them to the list.
new_user = {
    'last': 'fermi',
    'first': 'enrico',
    'username': 'efermi',
}
users.append(new_user)

# Make another new user, and add them as well.
new_user = {
    'last': 'curie',
    'first': 'marie',
    'username': 'mcurie',
}
users.append(new_user)

# Show all information about each user.
for user_dict in users:
    for k, v in user_dict.items():
        print(k + ": " + v)
    print("\n")
```

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

```
# Define a list of users, where each user
# is represented by a dictionary.
users = [
    {
        'last': 'fermi',
        'first': 'enrico',
        'username': 'efermi',
    },
    {
        'last': 'curie',
        'first': 'marie',
        'username': 'mcurie',
    },
]

# Show all information about each user.
for user_dict in users:
    for k, v in user_dict.items():
        print(k + ": " + v)
    print("\n")
```

Nesting — Lists in a dictionary

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

Storing lists in a dictionary

```
# Store multiple languages for each person.
fav_languages = {
    'jen': ['python', 'ruby'],
    'sarah': ['c'],
    'edward': ['ruby', 'go'],
    'phil': ['python', 'haskell'],
}

# Show all responses for each person.
for name, langs in fav_languages.items():
    print(name + ": ")
    for lang in langs:
        print("- " + lang)
```

Nesting — A dictionary of dictionaries

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

Storing dictionaries in a dictionary

```
users = {
    'aeinstein': {
        'first': 'albert',
        'last': 'einstein',
        'location': 'princeton',
    },
    'mcurie': {
        'first': 'marie',
        'last': 'curie',
        'location': 'paris',
    },
}

for username, user_dict in users.items():
    print("\nUsername: " + username)
    full_name = user_dict['first'] + " "
    full_name += user_dict['last']
    location = user_dict['location']

    print("\tFull name: " + full_name.title())
    print("\tLocation: " + location.title())
```

Levels of nesting

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

Using an OrderedDict

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

Preserving the order of keys and values

```
from collections import OrderedDict

# Store each person's languages, keeping
# track of who responded first.
fav_languages = OrderedDict()

fav_languages['jen'] = ['python', 'ruby']
fav_languages['sarah'] = ['c']
fav_languages['edward'] = ['ruby', 'go']
fav_languages['phil'] = ['python', 'haskell']

# Display the results, in the same order they
# were entered.
for name, langs in fav_languages.items():
    print(name + ":")
    for lang in langs:
        print("- " + lang)
```

Generating a million dictionaries

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

A million aliens

```
aliens = []

# Make a million green aliens, worth 5 points
# each. Have them all start in one row.
for alien_num in range(1000000):
    new_alien = {}
    new_alien['color'] = 'green'
    new_alien['points'] = 5
    new_alien['x'] = 20 * alien_num
    new_alien['y'] = 0
    aliens.append(new_alien)

# Prove the list contains a million aliens.
num_aliens = len(aliens)

print("Number of aliens created:")
print(num_aliens)
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — If Statements and While Loops

What are if statements? What are while loops?

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

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

Conditional Tests

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

Checking for equality

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

```
>>> car = 'bmw'
>>> car == 'bmw'
True
>>> car = 'audi'
>>> car == 'bmw'
False
```

Ignoring case when making a comparison

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
```

Checking for inequality

```
>>> topping = 'mushrooms'
>>> topping != 'anchovies'
True
```

Numerical comparisons

Testing numerical values is similar to testing string values.

Testing equality and inequality

```
>>> age = 18
>>> age == 18
True
>>> age != 18
False
```

Comparison operators

```
>>> age = 19
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

Checking multiple conditions

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

Using and to check multiple conditions

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 and age_1 >= 21
False
>>> age_1 = 23
>>> age_0 >= 21 and age_1 >= 21
True
```

Using or to check multiple conditions

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 or age_1 >= 21
True
>>> age_0 = 18
>>> age_0 >= 21 or age_1 >= 21
False
```

Boolean values

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

Simple boolean values

```
game_active = True
can_edit = False
```

If statements

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

Simple if statement

```
age = 19

if age >= 18:
    print("You're old enough to vote!")
```

If-else statements

```
age = 17

if age >= 18:
    print("You're old enough to vote!")
else:
    print("You can't vote yet.")
```

The if-elif-else chain

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 5
else:
    price = 10

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

Conditional tests with lists

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

Testing if a value is in a list

```
>>> players = ['al', 'bea', 'cyn', 'dale']
>>> 'al' in players
True
>>> 'eric' in players
False
```

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



Conditional tests with lists (cont.)

Testing if a value is not in a list

```
banned_users = ['ann', 'chad', 'dee']
user = 'erin'
```

```
if user not in banned_users:
    print("You can play!")
```

Checking if a list is empty

```
players = []
```

```
if players:
    for player in players:
        print("Player: " + player.title())
else:
    print("We have no players yet!")
```

Accepting input

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

Simple input

```
name = input("What's your name? ")
print("Hello, " + name + ".")
```

Accepting numerical input

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

if age >= 18:
    print("\nYou can vote!")
else:
    print("\nYou can't vote yet.")
```

Accepting input in Python 2.7

Use raw_input() in Python 2.7. This function interprets all input as a string, just as input() does in Python 3.

```
name = raw_input("What's your name? ")
print("Hello, " + name + ".")
```

While loops

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

Counting to 5

```
current_number = 1

while current_number <= 5:
    print(current_number)
    current_number += 1
```

While loops (cont.)

Letting the user choose when to quit

```
prompt = "\nTell me something, and I'll "
prompt += "repeat it back to you."
prompt += "\nEnter 'quit' to end the program. "
```

```
message = ""
while message != 'quit':
    message = input(prompt)

    if message != 'quit':
        print(message)
```

Using a flag

```
prompt = "\nTell me something, and I'll "
prompt += "repeat it back to you."
prompt += "\nEnter 'quit' to end the program. "
```

```
active = True
while active:
    message = input(prompt)
```

```
    if message == 'quit':
        active = False
    else:
        print(message)
```

Using break to exit a loop

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

```
while True:
    city = input(prompt)

    if city == 'quit':
        break
    else:
        print("I've been to " + city + "!")
```

Accepting input with Sublime Text

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.

Breaking out of loops

You can use the break statement and the continue statement with any of Python's loops. For example you can use break to quit a for loop that's working through a list or a dictionary. You can use continue to skip over certain items when looping through a list or dictionary as well.

While loops (cont.)

Using continue in a loop

```
banned_users = ['eve', 'fred', 'gary', 'helen']
```

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

```
players = []
while True:
    player = input(prompt)
    if player == 'quit':
        break
    elif player in banned_users:
        print(player + " is banned!")
        continue
    else:
        players.append(player)
```

```
print("\nYour team:")
for player in players:
    print(player)
```

Avoiding infinite loops

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

An infinite loop

```
while True:
    name = input("\nWho are you? ")
    print("Nice to meet you, " + name + "!")
```

Removing all instances of a value from a list

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

Removing all cats from a list of pets

```
pets = ['dog', 'cat', 'dog', 'fish', 'cat',
        'rabbit', 'cat']
print(pets)
```

```
while 'cat' in pets:
    pets.remove('cat')
```

```
print(pets)
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Functions

What are functions?

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

Defining a function

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

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

Making a function

```
def greet_user():  
    """Display a simple greeting."""  
    print("Hello!")  
  
greet_user()
```

Passing information to a function

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

Passing a single argument

```
def greet_user(username):  
    """Display a simple greeting."""  
    print("Hello, " + username + "!")  
  
greet_user('jesse')  
greet_user('diana')  
greet_user('brandon')
```

Positional and keyword arguments

The two main kinds of arguments are positional and keyword arguments. When you use positional arguments Python matches the first argument in the function call with the first parameter in the function definition, and so forth.

With keyword arguments, you specify which parameter each argument should be assigned to in the function call. When you use keyword arguments, the order of the arguments doesn't matter.

Using positional arguments

```
def describe_pet(animal, name):  
    """Display information about a pet."""  
    print("\nI have a " + animal + ".")  
    print("Its name is " + name + ".")  
  
describe_pet('hamster', 'harry')  
describe_pet('dog', 'willie')
```

Using keyword arguments

```
def describe_pet(animal, name):  
    """Display information about a pet."""  
    print("\nI have a " + animal + ".")  
    print("Its name is " + name + ".")  
  
describe_pet(animal='hamster', name='harry')  
describe_pet(name='willie', animal='dog')
```

Default values

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

Using a default value

```
def describe_pet(name, animal='dog'):  
    """Display information about a pet."""  
    print("\nI have a " + animal + ".")  
    print("Its name is " + name + ".")  
  
describe_pet('harry', 'hamster')  
describe_pet('willie')
```

Using None to make an argument optional

```
def describe_pet(animal, name=None):  
    """Display information about a pet."""  
    print("\nI have a " + animal + ".")  
    if name:  
        print("Its name is " + name + ".")  
  
describe_pet('hamster', 'harry')  
describe_pet('snake')
```

Return values

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

Returning a single value

```
def get_full_name(first, last):  
    """Return a neatly formatted full name."""  
    full_name = first + ' ' + last  
    return full_name.title()  
  
musician = get_full_name('jimi', 'hendrix')  
print(musician)
```

Returning a dictionary

```
def build_person(first, last):  
    """Return a dictionary of information  
    about a person."""  
    person = {'first': first, 'last': last}  
    return person  
  
musician = build_person('jimi', 'hendrix')  
print(musician)
```

Returning a dictionary with optional values

```
def build_person(first, last, age=None):  
    """Return a dictionary of information  
    about a person."""  
    person = {'first': first, 'last': last}  
    if age:  
        person['age'] = age  
    return person  
  
musician = build_person('jimi', 'hendrix', 27)  
print(musician)  
  
musician = build_person('janis', 'joplin')  
print(musician)
```

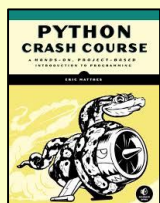
Visualizing functions

Try running some of these examples on pythontutor.com.

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



Passing a list to a function

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

Passing a list as an argument

```
def greet_users(names):
    """Print a simple greeting to everyone."""
    for name in names:
        msg = "Hello, " + name + "!"
        print(msg)

usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

Allowing a function to modify a list

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

```
def print_models(unprinted, printed):
    """3d print a set of models."""
    while unprinted:
        current_model = unprinted.pop()
        print("Printing " + current_model)
        printed.append(current_model)

# Store some unprinted designs,
# and print each of them.
unprinted = ['phone case', 'pendant', 'ring']
printed = []
print_models(unprinted, printed)

print("\nUnprinted:", unprinted)
print("Printed:", printed)
```

Preventing a function from modifying a list

The following example is the same as the previous one, except the original list is unchanged after calling print_models().

```
def print_models(unprinted, printed):
    """3d print a set of models."""
    while unprinted:
        current_model = unprinted.pop()
        print("Printing " + current_model)
        printed.append(current_model)

# Store some unprinted designs,
# and print each of them.
original = ['phone case', 'pendant', 'ring']
printed = []

print_models(original[:], printed)
print("\nOriginal:", original)
print("Printed:", printed)
```

Passing an arbitrary number of arguments

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

The ** operator allows a parameter to collect an arbitrary number of keyword arguments.

Collecting an arbitrary number of arguments

```
def make_pizza(size, *toppings):
    """Make a pizza."""
    print("\nMaking a " + size + " pizza.")
    print("Toppings:")
    for topping in toppings:
        print("- " + topping)

# Make three pizzas with different toppings.
make_pizza('small', 'pepperoni')
make_pizza('large', 'bacon bits', 'pineapple')
make_pizza('medium', 'mushrooms', 'peppers',
            'onions', 'extra cheese')
```

Collecting an arbitrary number of keyword arguments

```
def build_profile(first, last, **user_info):
    """Build a user's profile dictionary."""
    # Build a dict with the required keys.
    profile = {'first': first, 'last': last}

    # Add any other keys and values.
    for key, value in user_info.items():
        profile[key] = value

    return profile

# Create two users with different kinds
# of information.
user_0 = build_profile('albert', 'einstein',
                        location='princeton')
user_1 = build_profile('marie', 'curie',
                        location='paris', field='chemistry')

print(user_0)
print(user_1)
```

What's the best way to structure a function?

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

Modules

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

Storing a function in a module

File: pizza.py

```
def make_pizza(size, *toppings):
    """Make a pizza."""
    print("\nMaking a " + size + " pizza.")
    print("Toppings:")
    for topping in toppings:
        print("- " + topping)
```

Importing an entire module

File: making_pizzas.py

Every function in the module is available in the program file.

```
import pizza

pizza.make_pizza('medium', 'pepperoni')
pizza.make_pizza('small', 'bacon', 'pineapple')
```

Importing a specific function

Only the imported functions are available in the program file.

```
from pizza import make_pizza

make_pizza('medium', 'pepperoni')
make_pizza('small', 'bacon', 'pineapple')
```

Giving a module an alias

```
import pizza as p

p.make_pizza('medium', 'pepperoni')
p.make_pizza('small', 'bacon', 'pineapple')
```

Giving a function an alias

```
from pizza import make_pizza as mp

mp('medium', 'pepperoni')
mp('small', 'bacon', 'pineapple')
```

Importing all functions from a module

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

```
from pizza import *

make_pizza('medium', 'pepperoni')
make_pizza('small', 'bacon', 'pineapple')
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet - Classes

What are classes?

Classes are the foundation of object-oriented programming. Classes represent real-world things you want to model in your programs: for example dogs, cars, and robots. You use a class to make objects, which are specific instances of dogs, cars, and robots. A class defines the general behavior that a whole category of objects can have, and the information that can be associated with those objects.

Classes can inherit from each other – you can write a class that extends the functionality of an existing class. This allows you to code efficiently for a wide variety of situations.

Creating and using a 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 attributes, and the behavior is represented by functions. Functions that are part of a class are called methods.

The Car class

```
class Car():
    """A simple attempt to model a car."""

    def __init__(self, make, model, year):
        """Initialize car attributes."""
        self.make = make
        self.model = model
        self.year = year

    # Fuel capacity and level in gallons.
    self.fuel_capacity = 15
    self.fuel_level = 0

    def fill_tank(self):
        """Fill gas tank to capacity."""
        self.fuel_level = self.fuel_capacity
        print("Fuel tank is full.")

    def drive(self):
        """Simulate driving."""
        print("The car is moving.")
```

Creating and using a class (cont.)

Creating an object from a class

```
my_car = Car('audi', 'a4', 2016)
```

Accessing attribute values

```
print(my_car.make)
print(my_car.model)
print(my_car.year)
```

Calling methods

```
my_car.fill_tank()
my_car.drive()
```

Creating multiple objects

```
my_car = Car('audi', 'a4', 2016)
my_old_car = Car('subaru', 'outback', 2013)
my_truck = Car('toyota', 'tacoma', 2010)
```

Modifying attributes

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

Modifying an attribute directly

```
my_new_car = Car('audi', 'a4', 2016)
my_new_car.fuel_level = 5
```

Writing a method to update an attribute's value

```
def update_fuel_level(self, new_level):
    """Update the fuel level."""
    if new_level <= self.fuel_capacity:
        self.fuel_level = new_level
    else:
        print("The tank can't hold that much!")
```

Writing a method to increment an attribute's value

```
def add_fuel(self, amount):
    """Add fuel to the tank."""
    if (self.fuel_level + amount
        <= self.fuel_capacity):
        self.fuel_level += amount
        print("Added fuel.")
    else:
        print("The tank won't hold that much.")
```

Naming conventions

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.

Class inheritance

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

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

The __init__() method for a child class

```
class ElectricCar(Car):
    """A simple model of an electric car."""

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

    # Attributes specific to electric cars.
    # Battery capacity in kWh.
    self.battery_size = 70
    # Charge level in %.
    self.charge_level = 0
```

Adding new methods to the child class

```
class ElectricCar(Car):
    --snip--
    def charge(self):
        """Fully charge the vehicle."""
        self.charge_level = 100
        print("The vehicle is fully charged.")
```

Using child methods and parent methods

```
my_ecar = ElectricCar('tesla', 'model s', 2016)

my_ecar.charge()
my_ecar.drive()
```

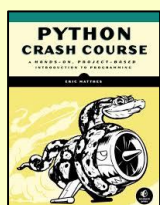
Finding your workflow

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.

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



Class inheritance (cont.)

Overriding parent methods

```
class ElectricCar(Car):
    --snip--
    def fill_tank(self):
        """Display an error message."""
        print("This car has no fuel tank!")
```

Instances as attributes

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

A Battery class

```
class Battery():
    """A battery for an electric car."""

    def __init__(self, size=70):
        """Initialize battery attributes."""
        # Capacity in kWh, charge level in %.
        self.size = size
        self.charge_level = 0

    def get_range(self):
        """Return the battery's range."""
        if self.size == 70:
            return 240
        elif self.size == 85:
            return 270
```

Using an instance as an attribute

```
class ElectricCar(Car):
    --snip--

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

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

    def charge(self):
        """Fully charge the vehicle."""
        self.battery.charge_level = 100
        print("The vehicle is fully charged.")
```

Using the instance

```
my_ecar = ElectricCar('tesla', 'model x', 2016)

my_ecar.charge()
print(my_ecar.battery.get_range())
my_ecar.drive()
```

Importing classes

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

Storing classes in a file

car.py

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

class Car():
    """A simple attempt to model a car."""
    --snip--

class Battery():
    """A battery for an electric car."""
    --snip--

class ElectricCar(Car):
    """A simple model of an electric car."""
    --snip--
```

Importing individual classes from a module

my_cars.py

```
from car import Car, ElectricCar

my_beetle = Car('volkswagen', 'beetle', 2016)
my_beetle.fill_tank()
my_beetle.drive()

my_tesla = ElectricCar('tesla', 'model s',
                        2016)
my_tesla.charge()
my_tesla.drive()
```

Importing an entire module

```
import car

my_beetle = car.Car(
    'volkswagen', 'beetle', 2016)
my_beetle.fill_tank()
my_beetle.drive()

my_tesla = car.ElectricCar(
    'tesla', 'model s', 2016)
my_tesla.charge()
my_tesla.drive()
```

Importing all classes from a module

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

```
from car import *

my_beetle = Car('volkswagen', 'beetle', 2016)
```

Classes in Python 2.7

Classes should inherit from object

```
class ClassName(object):
```

The Car class in Python 2.7

```
class Car(object):
```

Child class __init__() method is different

```
class ChildClassName(ParentClass):
    def __init__(self):
        super(ClassName, self).__init__()
```

The ElectricCar class in Python 2.7

```
class ElectricCar(Car):
    def __init__(self, make, model, year):
        super(ElectricCar, self).__init__(
            make, model, year)
```

Storing objects in a list

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

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

A fleet of rental cars

```
from car import Car, ElectricCar

# Make lists to hold a fleet of cars.
gas_fleet = []
electric_fleet = []

# Make 500 gas cars and 250 electric cars.
for _ in range(500):
    car = Car('ford', 'focus', 2016)
    gas_fleet.append(car)
for _ in range(250):
    ecar = ElectricCar('nissan', 'leaf', 2016)
    electric_fleet.append(ecar)

# Fill the gas cars, and charge electric cars.
for car in gas_fleet:
    car.fill_tank()
for ecar in electric_fleet:
    ecar.charge()

print("Gas cars:", len(gas_fleet))
print("Electric cars:", len(electric_fleet))
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Files and Exceptions

What are files? What are exceptions?

Your programs can read information in from files, and they can write data to files. Reading from files allows you to work with a wide variety of information; writing to files allows users to pick up where they left off the next time they run your program. You can write text to files, and you can store Python structures such as lists in data files.

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

Reading from a file

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

Reading an entire file at once

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    contents = f_obj.read()

print(contents)
```

Reading line by line

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

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    for line in f_obj:
        print(line.rstrip())
```

Reading from a file (cont.)

Storing the lines in a list

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    lines = f_obj.readlines()

for line in lines:
    print(line.rstrip())
```

Writing to a file

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

Writing to an empty file

```
filename = 'programming.txt'

with open(filename, 'w') as f:
    f.write("I love programming!")
```

Writing multiple lines to an empty file

```
filename = 'programming.txt'

with open(filename, 'w') as f:
    f.write("I love programming!\n")
    f.write("I love creating new games.\n")
```

Appending to a file

```
filename = 'programming.txt'

with open(filename, 'a') as f:
    f.write("I also love working with data.\n")
    f.write("I love making apps as well.\n")
```

File paths

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

Opening a file from a subfolder

```
f_path = "text_files/alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()

for line in lines:
    print(line.rstrip())
```

File paths (cont.)

Opening a file using an absolute path

```
f_path = "/home/ehmatthes/books/alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()
```

Opening a file on Windows

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

```
f_path = "C:\\Users\\ehmatthes\\books\\alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()
```

The try-except block

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

Handling the ZeroDivisionError exception

```
try:
    print(5/0)
except ZeroDivisionError:
    print("You can't divide by zero!")
```

Handling the FileNotFoundError exception

```
f_name = 'siddhartha.txt'

try:
    with open(f_name) as f_obj:
        lines = f_obj.readlines()
except FileNotFoundError:
    msg = "Can't find file {0}.".format(f_name)
    print(msg)
```

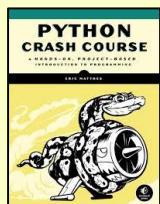
Knowing which exception to handle

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

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



The else block

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

Using an else block

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

x = input("First number: ")
y = input("Second number: ")

try:
    result = int(x) / int(y)
except ZeroDivisionError:
    print("You can't divide by zero!")
else:
    print(result)
```

Preventing crashes from user input

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

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

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

while True:
    x = input("\nFirst number: ")
    if x == 'q':
        break
    y = input("Second number: ")
    if y == 'q':
        break

    try:
        result = int(x) / int(y)
    except ZeroDivisionError:
        print("You can't divide by zero!")
    else:
        print(result)
```

Deciding which errors to report

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

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

Failing silently

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

Using the pass statement in an else block

```
f_names = ['alice.txt', 'siddhartha.txt',
           'moby_dick.txt', 'little_women.txt']

for f_name in f_names:
    # Report the length of each file found.
    try:
        with open(f_name) as f_obj:
            lines = f_obj.readlines()
    except FileNotFoundError:
        # Just move on to the next file.
        pass
    else:
        num_lines = len(lines)
        msg = "{0} has {1} lines.".format(
            f_name, num_lines)
        print(msg)
```

Avoid bare except blocks

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

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

Don't use bare except blocks

```
try:
    # Do something
except:
    pass
```

Use Exception instead

```
try:
    # Do something
except Exception:
    pass
```

Printing the exception

```
try:
    # Do something
except Exception as e:
    print(e, type(e))
```

Storing data with json

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

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

Using json.dump() to store data

```
"""Store some numbers."""

import json

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

filename = 'numbers.json'
with open(filename, 'w') as f_obj:
    json.dump(numbers, f_obj)
```

Using json.load() to read data

```
"""Load some previously stored numbers."""

import json

filename = 'numbers.json'
with open(filename) as f_obj:
    numbers = json.load(f_obj)

print(numbers)
```

Making sure the stored data exists

```
import json

f_name = 'numbers.json'

try:
    with open(f_name) as f_obj:
        numbers = json.load(f_obj)
except FileNotFoundError:
    msg = "Can't find {0}.".format(f_name)
    print(msg)
else:
    print(numbers)
```

Practice with exceptions

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

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Testing Your Code

Why test your code?

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

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

Testing a function: A passing test

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

A function to test

Save this as full_names.py

```
def get_full_name(first, last):
    """Return a full name."""
    full_name = "{0} {1}".format(first, last)
    return full_name.title()
```

Using the function

Save this as names.py

```
from full_names import get_full_name

janis = get_full_name('janis', 'joplin')
print(janis)

bob = get_full_name('bob', 'dylan')
print(bob)
```

Testing a function (cont.)

Building a testcase with one unit test

To build a test case, make a class that inherits from unittest.TestCase and write methods that begin with test_. Save this as test_full_names.py

```
import unittest
from full_names import get_full_name

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

    def test_first_last(self):
        """Test names like Janis Joplin."""
        full_name = get_full_name('janis',
                                   'joplin')
        self.assertEqual(full_name,
                           'Janis Joplin')

unittest.main()
```

Running the test

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

```
.
-----
Ran 1 test in 0.000s

OK
```

Testing a function: A failing test

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

Modifying the function

We'll modify get_full_name() so it handles middle names, but we'll do it in a way that breaks existing behavior.

```
def get_full_name(first, middle, last):
    """Return a full name."""
    full_name = "{0} {1} {2}".format(first,
                                       middle, last)

    return full_name.title()
```

Using the function

```
from full_names import get_full_name

john = get_full_name('john', 'lee', 'hooker')
print(john)

david = get_full_name('david', 'lee', 'roth')
print(david)
```

A failing test (cont.)

Running the test

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

```
E
=====
ERROR: test_first_last (__main__.NamesTestCase)
Test names like Janis Joplin.
-----
Traceback (most recent call last):
  File "test_full_names.py", line 10,
    in test_first_last
    'joplin')
TypeError: get_full_name() missing 1 required
positional argument: 'last'

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

FAILED (errors=1)

Fixing the code

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

```
def get_full_name(first, last, middle=''):
    """Return a full name."""
    if middle:
        full_name = "{0} {1} {2}".format(first,
                                           middle, last)
    else:
        full_name = "{0} {1}".format(first,
                                       last)

    return full_name.title()
```

Running the test

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

```
.
-----
Ran 1 test in 0.000s

OK
```

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



Adding new tests

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

Testing middle names

We've shown that `get_full_name()` works for first and last names. Let's test that it works for middle names as well.

```
import unittest
from full_names import get_full_name

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

    def test_first_last(self):
        """Test names like Janis Joplin."""
        full_name = get_full_name('janis',
                                   'joplin')
        self.assertEqual(full_name,
                           'Janis Joplin')

    def test_middle(self):
        """Test names like David Lee Roth."""
        full_name = get_full_name('david',
                                   'roth', 'lee')
        self.assertEqual(full_name,
                           'David Lee Roth')

unittest.main()
```

Running the tests

The two dots represent two passing tests.

```
..
-----
Ran 2 tests in 0.000s

OK
```

A variety of assert methods

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

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

```
assertEqual(a, b)
assertNotEqual(a, b)
```

Verify that `x` is True, or `x` is False

```
assertTrue(x)
assertFalse(x)
```

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

```
assertIn(item, list)
assertNotIn(item, list)
```

Testing a class

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

A class to test

Save as `accountant.py`

```
class Accountant():
    """Manage a bank account."""

    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount
```

Building a testcase

For the first test, we'll make sure we can start out with different initial balances. Save this as `test_accountant.py`.

```
import unittest
from accountant import Accountant

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

    def test_initial_balance(self):
        # Default balance should be 0.
        acc = Accountant()
        self.assertEqual(acc.balance, 0)

        # Test non-default balance.
        acc = Accountant(100)
        self.assertEqual(acc.balance, 100)

unittest.main()
```

Running the test

```
.
-----
Ran 1 test in 0.000s

OK
```

When is it okay to modify tests?

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

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

The `setUp()` method

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

Using `setUp()` to support multiple tests

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

```
import unittest
from accountant import Accountant

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

    def setUp(self):
        self.acc = Accountant()

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

        # Test non-default balance.
        acc = Accountant(100)
        self.assertEqual(acc.balance, 100)

    def test_deposit(self):
        # Test single deposit.
        self.acc.deposit(100)
        self.assertEqual(self.acc.balance, 100)

        # Test multiple deposits.
        self.acc.deposit(100)
        self.acc.deposit(100)
        self.assertEqual(self.acc.balance, 300)

    def test_withdrawal(self):
        # Test single withdrawal.
        self.acc.deposit(1000)
        self.acc.withdraw(100)
        self.assertEqual(self.acc.balance, 900)
```

```
unittest.main()
```

Running the tests

```
...
-----
Ran 3 tests in 0.001s

OK
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — matplotlib

What is matplotlib?

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

Installing matplotlib

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

matplotlib on Linux

```
$ sudo apt-get install python3-matplotlib
```

matplotlib on OS X

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

```
$ pip install --user matplotlib
```

matplotlib on Windows

You first need to install Visual Studio, which you can do from <https://dev.windows.com/>. The Community edition is free. Then go to <https://pypi.python.org/pypi/matplotlib/> or <http://www.lfd.uci.edu/~gohlke/pythonlibs/#matplotlib> and download an appropriate installer file.

Line graphs and scatter plots

Making a line graph

```
import matplotlib.pyplot as plt

x_values = [0, 1, 2, 3, 4, 5]
squares = [0, 1, 4, 9, 16, 25]
plt.plot(x_values, squares)
plt.show()
```

Line graphs and scatter plots (cont.)

Making a scatter plot

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

```
import matplotlib.pyplot as plt

x_values = list(range(1000))
squares = [x**2 for x in x_values]

plt.scatter(x_values, squares, s=10)
plt.show()
```

Customizing plots

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

Adding titles and labels, and scaling axes

```
import matplotlib.pyplot as plt

x_values = list(range(1000))
squares = [x**2 for x in x_values]
plt.scatter(x_values, squares, s=10)

plt.title("Square Numbers", fontsize=24)
plt.xlabel("Value", fontsize=18)
plt.ylabel("Square of Value", fontsize=18)
plt.tick_params(axis='both', which='major',
                labelsize=14)
plt.axis([0, 1100, 0, 1100000])

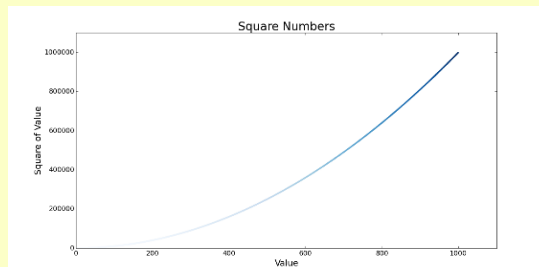
plt.show()
```

Using a colormap

A colormap varies the point colors from one shade to another, based on a certain value for each point. The value used to determine the color of each point is passed to the *c* argument, and the *cmap* argument specifies which colormap to use.

The `edgecolor='none'` argument removes the black outline from each point.

```
plt.scatter(x_values, squares, c=squares,
            cmap=plt.cm.Blues, edgecolor='none',
            s=10)
```



Customizing plots (cont.)

Emphasizing points

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

```
import matplotlib.pyplot as plt

x_values = list(range(1000))
squares = [x**2 for x in x_values]
plt.scatter(x_values, squares, c=squares,
            cmap=plt.cm.Blues, edgecolor='none',
            s=10)

plt.scatter(x_values[0], squares[0], c='green',
            edgecolor='none', s=100)
plt.scatter(x_values[-1], squares[-1], c='red',
            edgecolor='none', s=100)
```

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

Removing axes

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

```
plt.axes().get_xaxis().set_visible(False)
plt.axes().get_yaxis().set_visible(False)
```

Setting a custom figure size

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

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

Saving a plot

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

```
plt.savefig('squares.png', bbox_inches='tight')
```

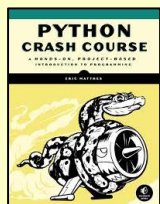
Online resources

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

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



Multiple plots

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

Plotting two sets of data

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

```
import matplotlib.pyplot as plt

x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

plt.scatter(x_values, squares, c='blue',
            edgecolor='none', s=20)
plt.scatter(x_values, cubes, c='red',
            edgecolor='none', s=20)
```

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

Filling the space between data sets

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

```
plt.fill_between(x_values, cubes, squares,
                 facecolor='blue', alpha=0.25)
```

Working with dates and times

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

Generating the current date

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

```
from datetime import datetime as dt

today = dt.now()
date_string = dt.strftime(today, '%m/%d/%Y')
print(date_string)
```

Generating a specific date

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

```
from datetime import datetime as dt

new_years = dt(2017, 1, 1)
fall_equinox = dt(year=2016, month=9, day=22)
```

Working with dates and times (cont.)

Datetime formatting arguments

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

%A	Weekday name, such as Monday
%B	Month name, such as January
%m	Month, as a number (01 to 12)
%d	Day of the month, as a number (01 to 31)
%Y	Four-digit year, such as 2016
%y	Two-digit year, such as 16
%H	Hour, in 24-hour format (00 to 23)
%I	Hour, in 12-hour format (01 to 12)
%p	AM or PM
%M	Minutes (00 to 59)
%S	Seconds (00 to 61)

Converting a string to a datetime object

```
new_years = dt.strptime('1/1/2017', '%m/%d/%Y')
```

Converting a datetime object to a string

```
ny_string = dt.strftime(new_years, '%B %d, %Y')
print(ny_string)
```

Plotting high temperatures

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

```
from datetime import datetime as dt

import matplotlib.pyplot as plt
from matplotlib import dates as mdates

dates = [
    dt(2016, 6, 21), dt(2016, 6, 22),
    dt(2016, 6, 23), dt(2016, 6, 24),
]

highs = [57, 68, 64, 59]

fig = plt.figure(dpi=128, figsize=(10,6))
plt.plot(dates, highs, c='red')
plt.title("Daily High Temps", fontsize=24)
plt.ylabel("Temp (F)", fontsize=16)

x_axis = plt.axes().get_xaxis()
x_axis.set_major_formatter(
    mdates.DateFormatter('%B %d %Y')
)
fig.autofmt_xdate()

plt.show()
```

Multiple plots in one figure

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

Sharing an x-axis

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

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

```
import matplotlib.pyplot as plt
```

```
x_vals = list(range(11))
squares = [x**2 for x in x_vals]
cubes = [x**3 for x in x_vals]
```

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

```
axarr[0].scatter(x_vals, squares)
axarr[0].set_title('Squares')
```

```
axarr[1].scatter(x_vals, cubes, c='red')
axarr[1].set_title('Cubes')
```

```
plt.show()
```

Sharing a y-axis

To share a y-axis, we use the `sharey=True` argument.

```
import matplotlib.pyplot as plt
```

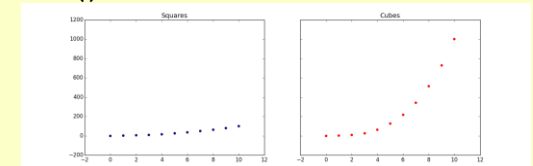
```
x_vals = list(range(11))
squares = [x**2 for x in x_vals]
cubes = [x**3 for x in x_vals]
```

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

```
axarr[0].scatter(x_vals, squares)
axarr[0].set_title('Squares')
```

```
axarr[1].scatter(x_vals, cubes, c='red')
axarr[1].set_title('Cubes')
```

```
plt.show()
```



More cheat sheets available at
ehmatthes.github.io/pcc/