# Beginner's Python Cheat Sheet

## Variables and Strings

*Variables are used to assign labels to values. A string is a series of characters, surrounded by single or double quotes. Python's f-strings allow you to use variables inside strings to build dynamic messages.*

### Hello world

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

### Hello world with a variable

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

### f-strings (using variables in strings)

```
first_name = 'albert'
last_name = 'einstein'
full_name = f"{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)
resolutions = ('720p', '1080p', '4K')
```

## If statements

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

### Conditional tests

```
equal              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 tests 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
elif age < 65:
    ticket_price = 40
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(f"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': 7, 'ever': 4, 'erin': 47}

for name, number in fav_numbers.items():
    print(f"{name} loves {number}.")
```

### Looping through all keys

```
fav_numbers = {'eric': 7, 'ever': 4, 'erin': 47}

for name in fav_numbers.keys():
    print(f"{name} loves a number.")
```

### Looping through all the values

```
fav_numbers = {'eric': 7, 'ever': 4, 'erin': 47}

for number in fav_numbers.values():
    print(f"{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(f"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)
```

## While loops

*A while loop repeats a block of code as long as a certain condition is true. While loops are especially useful when you can't know ahead of time how many times a loop should run.*

### 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(f"Hello, {username}!")

greet_user('jesse')
```

### Default values for parameters

```
def make_pizza(topping='pineapple'):
    """Make a single-topping pizza."""
    print(f"Have a {topping} pizza!")

make_pizza()
make_pizza('mushroom')
```

### 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(f"{self.name} is sitting.")

my_dog = Dog('Peso')

print(f"{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(f"{self.name} is searching.")

my_dog = SARDog('Willie')

print(f"{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. As you're learning you can write mall sprograms that relate to these ideas, so you can get practice writing code relevant to topics you're interested in.

## Working with files

*Your programs can read from files and write to files. Files are opened in read mode by default, but can also be opened in write mode and append mode.*

### 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

*The variable referring to the file object is often shortened to `f`.*

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

### Appending to a file

```
filename = 'journal.txt'
with open(filename, 'a') as f:
    f.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.

# 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 it clear that the variable represents more than one item.*

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 elements

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

## Modifying individual items

*Once you've defined a list, you can change the value of 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[1] = 'robert'
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. This allows you to modify existing lists, or start with an empty list and then add items to it as the program develops.*

Adding an element to the end of the list

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

Starting with an empty list

```
users = []
users.append('amy')
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 item. 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(f"We have {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 assigns it to 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(f"\nWelcome, {user}!")
    print("We're so glad you joined!")

print("\nWelcome, we're glad to see you all!")
```

## 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 statistical operations 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 subset 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 second 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*

Follow common Python formatting conventions:

- 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 usually designated by parentheses.*

*You can overwrite an entire tuple, but you can't change the values of individual elements.*

### 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)
print(dimensions)
```

## 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. Python Tutor 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(f"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_2e/

# 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 provided 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)
alien_speed = alien_0.get('speed')

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

## 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
```

### Starting with 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 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, and then run one of your own programs that uses dictionaries.*

## 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.*

*Dictionaries keep track of the order in which key-value pairs are added. If you want to process the information in a different order, you can sort the keys in your loop, using the `sorted()` function.*

### 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(f"{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 reverse order

```
# Show each person's favorite language,
#   in reverse order by the person's name.
for name in sorted(fav_languages.keys(),
        reverse=True):
    language = fav_languages[name]
    print(f"{name}: {language}")
```

## Dictionary length

*You can find the number of key-value pairs in a dictionary using the `len()` function.*

### Finding a dictionary's length

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

## Nesting - A list of dictionaries

*It's sometimes useful to store a number 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.
print("User summary:")
for user_dict in users:
    for k, v in user_dict.items():
        print(f"{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.
print("User summary:")
for user_dict in users:
    for k, v in user_dict.items():
        print(f"{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(f"{name}: ")
    for lang in langs:
        print(f"- {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():
    full_name = f"{user_dict['first']} "
    full_name += user_dict['last']

    location = user_dict['location']

    print("\nUsername: " + username)
    print(f"\tFull name: {full_name.title()}")
    print(f"\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.*

## Dictionary Comprehensions

*A comprehension is a compact way of generating a dictionary, similar to a list comprehension. To make a dictionary comprehension, define an expression for the key-value pairs you want to make. Then write a for loop to generate the values that will feed into this expression.*

*The `zip()` function matches each item in one list to each item in a second list. It can be used to make a dictionary from two lists.*

### Using a loop to make a dictionary

```
squares = {}
for x in range(5):
    squares[x] = x**2
```

### Using a dictionary comprehension

```
squares = {x:x**2 for x in range(5)}
```

### Using `zip()` to make a dictionary

```
group_1 = ['kai', 'abe', 'ada', 'gus', 'zoe']
group_2 = ['jen', 'eva', 'dan', 'isa', 'meg']

pairings = {name:name_2
    for name, name_2 in zip(group_1, group_2)}
```

## 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(1_000_000):
    new_alien = {
        'color': 'green',
        'points': 5,
        'x': 20 * alien_num,
        '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_2e/

# Beginner's Python Cheat Sheet - If Statements and While Loops

## What are if statements? What are while loops?

Python's 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 statements that identify the exact conditions you're interested in.

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.*

*If your conditional tests aren't doing what you expect them to, make sure you're not accidentally using a single equal sign.*

```
>>> 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
is_valid = 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 = 25
else:
    price = 40

print(f"Your cost is ${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
```

### Testing if two values are in a list

```
>>> 'al' in players and 'cyn' in players
```

## 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

*An empty list evaluates as `False` in an if statement.*

```
players = []

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

## Accepting input

*You can allow your users to enter input using the `input()` function. All input is initially stored as a string. If you want to accept numerical input, you'll need to convert the input string value to a numerical type.*

### Simple input

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

### Accepting numerical input using int()

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

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

### Accepting numerical input using float()

```
tip = input("How much do you want to tip? ")
tip = float(tip)
print(f"Tipped ${tip}.")
```

## 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

*Flags are most useful in long-running programs where code from other parts of the program might need to end the loop.*

```
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(f"I've been to {city}!")
```

## Accepting input with Sublime Text

*Sublime Text, and a number of other text editors can't run programs that prompt the user for input. You can use these editors to write programs that prompt for input, but you'll need to run them 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(f"{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. You can usually press Ctrl-C to stop an infinite loop.*

### An infinite loop

```
while True:
    name = input("\nWho are you? ")
    print(f"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_2e/

# 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 simple argument

```
def greet_user(username):
    """Display a simple greeting."""
    print(f"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(f"\nI have a {animal}.")
    print(f"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(f"\nI have a {animal}.")
    print(f"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(f"\nI have a {animal}.")
    print(f"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(f"\nI have a {animal}.")
    if name:
        print(f"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 should provide a variable which the return value can be assigned to. 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 = f"{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, and some of your own programs that use functions, on pythontutor.com.*

## 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 = f"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 first 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(f"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(f"\nUnprinted: {unprinted}")
print(f"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(f"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(f"\nOriginal: {original}")
print(f"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. This parameter is often named* `*args`.

*The ** operator allows a parameter to collect an arbitrary number of keyword arguments. These are stored as a dictionary with the parameter names as keys, and the arguments as values. This is often named* `**kwargs`.

### Collecting an arbitrary number of arguments

```
def make_pizza(size, *toppings):
    """Make a pizza."""
    print(f"\nMaking a {size} pizza.")

    print("Toppings:")
    for topping in toppings:
        print(f"- {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 dictionary for a user."""
    user_info['first'] = first
    user_info['last'] = last

    return user_info

# 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?

*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 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(f"\nMaking a {size} pizza.")
    print("Toppings:")
    for topping in toppings:
        print(f"- {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_2e/

# 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 such as 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. Even if you don't write many of your own classes, you'll frequently find yourself working with classes that others have written.

## 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 assigned to 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 instance from a class

```
my_car = Car('audi', 'a4', 2021)
```

### 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 instances

```
my_car = Car('audi', 'a4', 2021)
my_old_car = Car('subaru', 'outback', 2015)
my_truck = Car('toyota', 'tacoma', 2018)
my_old_truck = Car('ford', 'ranger', 1999)
```

## Modifying attributes

*You can modify an attribute's value directly, or you can write methods that manage updating values more carefully. Methods like these can help validate the kinds of changes that are being made to an attribute.*

### Modifying an attribute directly

```
my_new_car = Car('audi', 'a4', 2021)
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 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 = 85

        # 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', 2021)

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.*

## 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 more complex real-world things and concepts.*

### A Battery class

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

    def __init__(self, size=85):
        """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 == 85:
            return 390
        elif self.size == 100:
            return 415
```

### 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 s', 2021)

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', 2019)
my_beetle.fill_tank()
my_beetle.drive()

my_tesla = ElectricCar('tesla', 'model s',
        2021)
my_tesla.charge()
my_tesla.drive()
```

### Importing an entire module

```
import car

my_beetle = car.Car(
        'volkswagen', 'beetle', 2019)
my_beetle.fill_tank()
my_beetle.drive()

my_tesla = car.ElectricCar(
        'tesla', 'model s', 2021)
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', 2019)
my_tesla = ElectricCar('tesla', 'model s',
        2021)
```

## 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 250 gas cars and 500 electric cars.
for _ in range(250):
    car = Car('ford', 'escape', 2021)
    gas_fleet.append(car)
for _ in range(500):
    ecar = ElectricCar('nissan', 'leaf', 2021)
    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(f"Gas cars: {len(gas_fleet)}")
print(f"Electric cars: {len(electric_fleet)}")
```

## Understanding self

*People often ask what the `self` variable represents. The `self` variable is a reference to an object that's been created from the class.*

*The `self` variable provides a way to make other variables and objects available everywhere in a class. The `self` variable is automatically passed to each method that's called through an object, which is why you see it listed first in every method definition. Any variable attached to `self` is available everywhere in the class.*

## Understanding `__init__()`

*The `__init__()` method is a function that's part of a class, just like any other method. The only special thing about `__init__()` is that it's called automatically every time you make a new instance from a class. If you accidentally misspell `__init__()`, the method won't be called and your object may not be created correctly.*

*More cheat sheets available at*

ehmatthes.github.io/pcc_2e/

# 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 it line by line. The `with` statement shown here 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 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:
    lines = f.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:
    lines = f.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:
    lines = f.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:
        lines = f.readlines()
except FileNotFoundError:
    msg = f"Can't find file: {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. It's a good idea to skim through the exceptions listed at docs.python.org/3/library/exceptions.html.*

## 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 caused by 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 `except` 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:
            lines = f.readlines()
    except FileNotFoundError:
        # Just move on to the next file.
        pass
    else:
        num_lines = len(lines)
        msg = f"{f_name} has {num_lines}"
        msg += " 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:
    json.dump(numbers, f)
```

### Using `json.load()` to read data

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

import json

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

print(numbers)
```

### Making sure the stored data exists

```
import json

f_name = 'numbers.json'
try:
    with open(f_name) as f:
        numbers = json.load(f)
except FileNotFoundError:
    msg = f"Can't find file: {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. Run your program with appropriate and inappropriate data, and make sure it handles each situation correctly.*

*More cheat sheets available at*

ehmatthes.github.io/pcc_2e/

# 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 whether or not you've broken existing behavior by running your tests.

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 that your code's behavior is correct 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 = f"{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')

if __name__ == '__main__':
    unittest.main()
```

### Running the test

*Python reports on each unit test in the test case. The dot represents 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 = f"{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, otherwise your code will break for anyone who's using it the same way it's being used in the failing test. Here we can make the middle name optional.*

```
def get_full_name(first, last, middle=''):
    """Return a full name."""

    if middle:
        full_name = f"{first} {middle} {last}"
    else:
        full_name = f"{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
```

## 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')

if __name__ == '__main__':
    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, and no one is using your code yet.*

## 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)

if __name__ == '__main__':
    unittest.main()
```

### Running the tests

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

OK
```

*More cheat sheets available at*

ehmatthes.github.io/pcc_2e/

# Beginner's Python Cheat Sheet - Pygame

## What is Pygame?

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.

## Installing Pygame

*Pygame runs on all systems, and you should be able to install it in one line.*

### Installing Pygame

```
$ python -m pip install --user pygame
```

## Starting a game

*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

class AlienInvasion:

    def __init__(self):
        pygame.init()
        self.screen = pygame.display.set_mode(
                (1200, 800))
        pygame.display.set_caption(
                "Alien Invasion")

    def run_game(self):
        while True:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    sys.exit()

            pygame.display.flip()

if __name__ == '__main__':
    ai = AlienInvasion()
    ai.run_game()
```

## Starting a game (cont.)

### Setting a custom window size

*The `display.set_mode()` function accepts a tuple that defines the screen size.*

```
screen_dim = (1500, 1000)
self.screen = pygame.display.set_mode(
        screen_dim)
```

### Setting a custom background color

*Colors are defined as a tuple of red, green, and blue values. Each value ranges from 0-255. The `fill()` method fills the screen with the color you specify, and should be called before you add any other elements to the screen.*

```
def __init__(self):
    --snip--
    self.bg_color = (225, 225, 225)

def run_game(self):
    while True:
        for event in pygame.event.get():
            --snip--

        self.screen.fill(self.bg_color)
        pygame.display.flip()
```

## Pygame rect objects

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

### Getting the screen rect object

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

```
self.screen_rect = self.screen.get_rect()
```

### Finding the center of the screen

*Rect objects have a `center` attribute which stores the center point.*

```
screen_center = self.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. The `self` variable has been left off for clarity.)*

```
# 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
```

## Pygame rect objects (cont.)

### 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 = pygame.Rect(100, 100, 3, 15)
color = (100, 100, 100)

pygame.draw.rect(screen, color, bullet_rect)
```

## Working with images

*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 = pygame.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 at the bottom center of the screen, by matching the `midbottom` of the ship with the `midbottom` of the screen.*

```
ship_rect.midbottom = screen_rect.midbottom
```

### Drawing an image to the screen

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

```
# Draw ship to screen.
screen.blit(ship, ship_rect)
```

### Transforming an image

*The `transform` module allows you to make changes to an image such as rotation and scaling.*

```
rotated_image = pygame.transform.rotate(
        ship.image, 45)
```

## Working with images (cont.)

### The `blitme()` method

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

```
def blitme(self):
    """Draw ship at current location."""
    self.screen.blit(self.image, self.rect)
```

## Responding to keyboard input

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

### Responding to key presses

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

```
for event in pygame.event.get():
    if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_RIGHT:
            ship_rect.x += 1
        elif event.key == pygame.K_LEFT:
            ship_rect.x -= 1
        elif event.key == pygame.K_SPACE:
            ship.fire_bullet()
        elif event.key == pygame.K_q:
            sys.exit()
```

### Responding to released keys

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

```
for event in pygame.event.get():
    if event.type == pygame.KEYUP:
        if event.key == pygame.K_RIGHT:
            ship.moving_right = False
```

## The game is an object

*In the overall structure shown here (under Starting a Game), the entire game is written as a class. This makes it possible to write programs that play the game automatically, and it also means you can build an arcade with a collection of games.*

## Pygame documentation

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

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

## Responding to mouse events

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

### Responding to the mouse button

```
for event in pygame.event.get():
    if event.type == pygame.MOUSEBUTTONDOWN:
        ship.fire_bullet()
```

### Finding the mouse position

*The mouse position is returned as a tuple.*

```
mouse_pos = pygame.mouse.get_pos()
```

### Clicking a button

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

```
if button_rect.collidepoint(mouse_pos):
    start_game()
```

### Hiding the mouse

```
pygame.mouse.set_visible(False)
```

## Pygame groups

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

### Making and filling a group

*An object that will be placed in a group must inherit from `Sprite`.*

```
from pygame.sprite import Sprite, Group

class Bullet(Sprite):
    ...
    def draw_bullet(self):
        ...
    def update(self):
        ...

bullets = Group()

new_bullet = Bullet()
bullets.add(new_bullet)
```

### Looping through the items in a group

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

```
for bullet in bullets.sprites():
    bullet.draw_bullet()
```

### Calling update() on a group

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

```
bullets.update()
```

## Pygame groups (cont.)

### Removing an item from a group

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

```
bullets.remove(bullet)
```

## Detecting collisions

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

### Collisions between a single object and a group

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

```
if pygame.sprite.spritecollideany(ship, aliens):
    ships_left -= 1
```

### Collisions between two groups

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

```
collisions = pygame.sprite.groupcollide(
        bullets, aliens, True, True)

score += len(collisions) * alien_point_value
```

## Rendering text

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

### Displaying a message

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

```
msg = "Play again?"
msg_color = (100, 100, 100)
bg_color = (230, 230, 230)

f = pygame.font.SysFont(None, 48)
msg_image = f.render(msg, True, msg_color,
        bg_color)
msg_image_rect = msg_image.get_rect()
msg_image_rect.center = screen_rect.center
screen.blit(msg_image, msg_image_rect)
```

*More cheat sheets available at*

ehmatthes.github.io/pcc_2e/

# Beginner's Python Cheat Sheet - Matplotlib

## What is Matplotlib?

Data visualization involves exploring data through visual representations. The Matplotlib library 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, and you should be able to install it in one line.*

### Installing Matplotlib

```
$ python -m pip install --user matplotlib
```

## Line graphs and scatter plots

### Making a line graph

*The `fig` object represents the entire figure, or collection of plots; `ax` represents a single plot in the figure. This convention is used even when there's only one plot in the figure.*

```
import matplotlib.pyplot as plt

x_values = [0, 1, 2, 3, 4, 5]
squares = [0, 1, 4, 9, 16, 25]

fig, ax = plt.subplots()
ax.plot(x_values, squares)

plt.show()
```

### Making a scatter plot

`scatter()` *takes a list of x and y values; 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]

fig, ax = plt.subplots()
ax.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 modified.*

### Using built-in styles

*Matplotlib comes with a number of built-in styles, which you can use with one additional line of code. The style must be specified before you create the figure.*

```
import matplotlib.pyplot as plt

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

plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.scatter(x_values, squares, s=10)

plt.show()
```

### Seeing available styles

*You can see all available styles on your system. This can be done in a terminal session.*

```
>>> import matplotlib.pyplot as plt
>>> plt.style.available
['seaborn-dark', 'seaborn-darkgrid', ...
```

### 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]

# Set overall style to use, and plot data.
plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.scatter(x_values, squares, s=10)

# Set chart title and label axes.
ax.set_title('Square Numbers', fontsize=24)
ax.set_xlabel('Value', fontsize=14)
ax.set_ylabel('Square of Value', fontsize=14)

# Set scale of axes, and size of tick labels.
ax.axis([0, 1100, 0, 1_100_000])
ax.tick_params(axis='both', labelsize=14)

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.*

```
ax.scatter(x_values, squares, c=squares,
        cmap=plt.cm.Blues, s=10)
```

## Customizing plots (cont.)

### Emphasizing points

*You can plot as much data as you want on one plot. Here we replot 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]

fig, ax = plt.subplots()
ax.scatter(x_values, squares, c=squares,
        cmap=plt.cm.Blues, s=10)

ax.scatter(x_values[0], squares[0], c='green',
        s=100)
ax.scatter(x_values[-1], squares[-1], c='red',
        s=100)

ax.set_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.*

```
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)
```

### Setting a custom figure size

*You can make your plot as big or small as you want by using the* `figsize` *argument. 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.*

```
fig, ax = plt.subplots(figsize=(10, 6),
        dpi=128)
```

### Saving a plot

*The Matplotlib viewer has a save button, but you can also save your visualizations programmatically by replacing* `plt.show()` *with* `plt.savefig()`*. The* `bbox_inches` *argument reduces the amount of whitespace around the figure.*

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

## Online resources

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

# 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 `ax.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.style.use('seaborn')
fig, ax = plt.subplots()

ax.scatter(x_values, squares, c='blue', s=10)
ax.scatter(x_values, cubes, c='red', s=10)

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.*

```
ax.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 = today.strftime('%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(2021, 1, 1)
fall_equinox = dt(year=2021, month=9, day=22)
```

# Working with dates and times (cont.)

## Datetime formatting arguments

*The `strptime()` function generates a `datetime` object from a string, and the `strftime()` method generates a formatted string from a `datetime` object. 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 2021
%y    Two-digit year, such as 21
%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('1/1/2021', '%m/%d/%Y')
```

## Converting a datetime object to a string

```
ny_string = new_years.strftime('%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(2020, 6, 21), dt(2020, 6, 22),
    dt(2020, 6, 23), dt(2020, 6, 24),
    ]

highs = [56, 57, 57, 64]

fig, ax = plt.subplots()
ax.plot(dates, highs, c='red')

ax.set_title("Daily High Temps", fontsize=24)
ax.set_ylabel("Temp (F)", fontsize=16)
x_axis = ax.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.*

## Sharing an x-axis

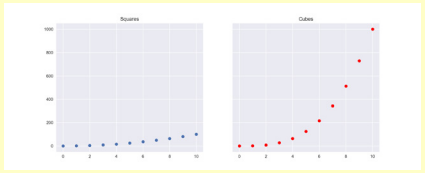
*The following code plots a set of squares and a set of cubes on two separate graphs that share a common x-axis. The `plt.subplots()` function returns a figure object and a tuple of axes. Each set of axes corresponds to a separate plot in the figure. The first two arguments control the number of rows and columns generated in the figure.*

```
import matplotlib.pyplot as plt

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

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

axs[0].scatter(x_values, squares)
axs[0].set_title('Squares')

axs[1].scatter(x_values, cubes, c='red')
axs[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_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

plt.style.use('seaborn')
fig, axs = plt.subplots(1, 2, sharey=True)

axs[0].scatter(x_values, squares)
axs[0].set_title('Squares')

axs[1].scatter(x_values, cubes, c='red')
axs[1].set_title('Cubes')

plt.show()
```



*More cheat sheets available at*

ehmatthes.github.io/pcc_2e/

# Beginner's Python Cheat Sheet - Plotly

## What is Plotly?

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

## Installing Plotly

*Plotly runs on all systems, and can be installed in one line.*

### Installing Plotly

```
$ python -m pip install --user plotly
```

## Line graphs, scatter plots, and bar graphs

*To make a plot with Plotly, you specify the data and then pass it to a graph object. The data is stored in a list, so you can add as much data as you want to any graph. In offline mode, the output should open automatically in a browser window.*

### Making a line graph

*A line graph is a scatter plot where the points are connected. Plotly generates JavaScript code to render the plot file. If you're curious to see the code, open the squares.html file in a text editor after running this program.*

```
from plotly.graph_objs import Scatter
from plotly import offline

# Define the data.
x_values = list(range(11))
squares = [x**2 for x in x_values]

# Pass the data to a graph object, and store it
#   in a list.
data = [Scatter(x=x_values, y=squares)]

# Pass the data and a filename to plot().
offline.plot(data, filename='squares.html')
```

### Making a scatter plot

*To make a scatter plot, use the `mode='markers'` argument to tell Plotly to only display the markers.*

```
data = [Scatter(x=x_values, y=squares,
        mode='markers')]
```

## Line graphs, scatter plots, and bar graphs (cont.)

### Making a bar graph

*To make a bar graph, pass your data to the `Bar()` graph object.*

```
from plotly.graph_objs import Bar
--snip--

data = [Bar(x=x_values, y=squares)]

# Pass the data and a filename to plot().
offline.plot(data, filename='squares.html')
```

## Adding a title and labels

### Using layout objects

*The `Layout` class allows you to specify titles, labels, and other formatting directives for your visualizations.*

```
from plotly.graph_objs import Scatter, Layout
from plotly import offline

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

data = [Scatter(x=x_values, y=squares)]

# Add a title, and a label for each axis.
title = 'Square Numbers'
x_axis_config = {'title': 'x'}
y_axis_config = {'title': 'Square of x'}

my_layout = Layout(title=title,
    xaxis=x_axis_config, yaxis=y_axis_config)

offline.plot(
        {'data': data, 'layout': my_layout},
        filename='squares.html')
```

## Specifying complex data

### Data as a dictionary

*Plotly is highly customizable, and most of that flexibility comes from representing data and formatting directives as a dictionary. Here is the same data from the previous examples, defined as a dictionary.*

*Defining the data as a dictionary also allows you to specify more information about each series. Anything that pertains to a specific data series such as markers, lines, and point labels, goes in the data dictionary. Plotly has several ways of specifying data, but internally all data is represented in this way.*

```
data = [{
    'type': 'scatter',
    'x': x_values,
    'y': squares,
    'mode': 'markers',
}]
```

## Multiple plots

*You can include as many data series as you want in a visualization. To do this, create one dictionary for each data series, and put these dictionaries in the data list. Each of these dictionaries is referred to as a trace in the Plotly documentation.*

### Plotting squares and cubes

*Here we use the `'name'` attribute to set the label for each trace.*

```
from plotly.graph_objs import Scatter
from plotly import offline

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

data = [
    {
        # Trace 1: squares
        'type': 'scatter',
        'x': x_values,
        'y': squares,
        'name': 'Squares',
    },
    {
        # Trace 2: cubes
        'type': 'scatter',
        'x': x_values,
        'y': cubes,
        'name': 'Cubes',
    },
]

offline.plot(data,
        filename='squares_cubes.html')
```

## Online resources

*The Plotly documentation is extensive and well-organized. Start with the overview at plotly.com/python/. Here you can see an example of all the basic chart types, and click on any example to see a relevant tutorial.*

*Then take a look at the Python Figure Reference, at plotly.com/python/reference/. Check out the Figure Data Structure in Python page as well, at plotly.com/python/figure-structure/.*

## Specifying complex layouts

*You can also specify the layout of your visualization as a dictionary, which gives you much more control of the overall layout.*

### Layout as a dictionary

*Here is the same layout we used earlier, written as a dictionary. Simple elements such as the title of the chart are just key-value pairs. More complex elements such as axes, which can have many of their own settings, are nested dictionaries.*

```
my_layout = {
    'title': 'Square Numbers',
    'xaxis': {
        'title': 'x',
    },
    'yaxis': {
        'title': 'Square of x',
    },
}
```

### A more complex layout

*Here is a layout for the same data, with more specific formatting directives in the data and layout dictionaries.*

```
from plotly.graph_objs import Scatter
from plotly import offline

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

data = [{
    'type': 'scatter',
    'x': x_values,
    'y': squares,
    'mode': 'markers',
    'marker': {
        'size': 10,
        'color': '#6688dd',
    },
}]

my_layout = {
    'title': 'Square Numbers',
    'xaxis': {
        'title': 'x',
        'titlefont': {'family': 'monospace'},
    },
    'yaxis': {
        'title': 'Square of x',
        'titlefont': {'family': 'monospace'},
    },
}

offline.plot(
       {'data': data, 'layout': my_layout},
       filename='squares.html')
```

## Specifying complex layouts (cont.)

### Using a colorscale

*Colorscales are often used to show variations in large datasets. In Plotly, colorscales are set in the marker dictionary, nested inside a data dictionary.*

```
data = [{
    'type': 'scatter',
    'x': x_values,
    'y': squares,
    'mode': 'markers',
    'marker': {
        'colorscale': 'Viridis',
        'color': squares,
        'colorbar': {'title': 'Value'},
    },
}]
```

## Using Subplots

*It's often useful to have multiple plots share the same axes. This is done using the subplots module.*

### Adding subplots to a figure

*To use the* `subplots` *module, make a figure to hold all the charts that will be made. Then use the* `add_trace()` *method to add each data series to the overall figure.*

*For more help, see the documentation at plot.ly/python/subplots/.*

```
from plotly.subplots import make_subplots
from plotly.graph_objects import Scatter
from plotly import offline

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

# Make two subplots, sharing a y-axis.
fig = make_subplots(rows=1, cols=2,
        shared_yaxes=True)

data = {
    'type': 'scatter',
    'x': x_values,
    'y': squares,
}
fig.add_trace(data, row=1, col=1)

data = {
    'type': 'scatter',
    'x': x_values,
    'y': cubes,
}
fig.add_trace(data, row=1, col=2)

offline.plot(fig, filename='subplots.html')
```

## Plotting global datasets

*Plotly has a variety of mapping tools. For example, if you have a set of points represented by latitude and longitude, you can create a scatter plot of those points overlaying a map.*

### The scattergeo chart type

*Here's a map showing the location of three of the higher peaks in North America. If you hover over each point, you'll see its location and the name of the mountain.*

```
from plotly import offline

# Points in (lat, lon) format.
peak_coords = [
    (63.069, -151.0063),
    (60.5671, -140.4055),
    (46.8529, -121.7604),
]

# Make matching lists of lats, lons,
#   and labels.
lats = [pc[0] for pc in peak_coords]
lons = [pc[1] for pc in peak_coords]
peak_names = ['Denali', 'Mt Logan',
        'Mt Rainier']

data = [{
    'type': 'scattergeo',
    'lon': lons,
    'lat': lats,
    'marker': {
        'size': 20,
        'color': '#227722',
    },
    'text': peak_names,
}]

my_layout = {
    'title': 'Selected High Peaks',
    'geo': {
        'scope': 'north america',
        'showland': True,
        'showocean': True,
        'showlakes': True,
        'showrivers': True,
    },
}

offline.plot(
        {'data': data, 'layout': my_layout},
        filename='peaks.html')
```

*More cheat sheets available at*

ehmatthes.github.io/pcc_2e/

# Beginner's Python Cheat Sheet - Django

## What is Django?

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

Django works well for tiny projects, and just as well for sites with millions of users.

## Installing Django

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

### Create a virtual environment

```
$ python -m venv ll_env
```

### Activate the environment (macOS and Linux)

```
$ source ll_env/bin/activate
```

### Activate the environment (Windows)

```
> ll_env\Scripts\activate
```

### Install Django to the active environment

```
(ll_env)$ pip install Django
```

## Creating a project

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

### Create a new project

*Make sure to include the dot at the end of this command.*

```
$ django-admin startproject learning_log .
```

### Create a database

```
$ python manage.py migrate
```

### View the project

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

```
$ python manage.py runserver
```

### Create a new app

*A Django project is made up of one or more apps.*

```
$ python manage.py startapp learning_logs
```

## Working with models

*The data in a Django project is structured as a set of models. Each model is represented by a class.*

### Defining a model

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

```
from django.db import models

class Topic(models.Model):
    """A topic the user is learning about."""

    text = models.CharField(max_length=200)
    date_added = models.DateTimeField(
            auto_now_add=True)

    def __str__(self):
        return self.text
```

### Activating a model

*To use a model the app must be added to the list `INSTALLED_APPS`, which is stored in the project's settings.py file.*

```
INSTALLED_APPS = [
    # My apps.
    'learning_logs',

    # Default Django apps.
    'django.contrib.admin',
]
```

### Migrating the database

*The database needs to be modified to store the kind of data that the model represents. You'll need to run these commands every time you create a new model, or modify an existing model.*

```
$ python manage.py makemigrations learning_logs
$ python manage.py migrate
```

### Creating a superuser

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

```
$ python manage.py createsuperuser
```

### Registering a model

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

```
from django.contrib import admin

from .models import Topic

admin.site.register(Topic)
```

## Building a simple home page

*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.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('learning_logs.urls')),
]
```

### 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.urls import path

from . import views

app_name = 'learning_logs'
urlpatterns = [
    # Home page.
    path('', 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')
```

## Online resources

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

## Building a simple home page (cont.)

### Writing a simple template

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

*The home page template will be saved as learning_logs/templates/learning_logs/index.html.*

```
<p>Learning Log</p>

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

## Template Inheritance

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

### The parent template

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

```
<p>
  <a href="{% url 'learning_logs:index' %}">
    Learning Log
  </a>
</p>

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

### The child template

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

```
{% extends 'learning_logs/base.html' %}

{% block content %}

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

{% endblock content %}
```

## Template indentation

*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.*

## Another model

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

### Defining a model with a foreign key

```
class Entry(models.Model):
    """Learning log entries for a topic."""
    topic = models.ForeignKey(Topic,
            on_delete=models.CASCADE)
    text = models.TextField()
    date_added = models.DateTimeField(
            auto_now_add=True)

    def __str__(self):
        return f"{self.text[:50]}..."
```

## Building a page with data

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

### URL parameters

*A URL often needs to accept a parameter telling it what data to access from the database. The URL pattern shown here looks for the ID of a specific topic and assigns it to the parameter* `topic_id`.

```
urlpatterns = [
    --snip--
    # Detail page for a single topic.
    path('topics/<int:topic_id>/', views.topic,
            name='topic'),
]
```

### Using data in a view

*The view uses a parameter from the URL to pull the correct data from the database. In this example the view is sending a* `context` *dictionary to the template, containing data that should be displayed on the page. You'll need to import any model you're using.*

```
def topic(request, topic_id):
    """Show a topic and all its entries."""
    topic = Topic.objects.get(id=topic_id)
    entries = topic.entry_set.order_by(
            '-date_added')
    context = {
        'topic': topic,
        'entries': entries,
    }
    return render(request,
            'learning_logs/topic.html', context)
```

## Restarting the development server

*If you make a change to your project and the change doesn't seem to have any effect, try restarting the server:*

*$ python manage.py runserver*

## Building a page with data (cont.)

### Using data in a template

*The data in the view function's* `context` *dictionary is available within the template. This data is accessed using template variables, which are indicated by doubled curly braces.*

*The vertical line after a template variable indicates a filter. In this case a filter called* `date` *formats date objects, and the filter* `linebreaks` *renders paragraphs properly on a web page.*

```
{% extends 'learning_logs/base.html' %}

{% block content %}

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

  <p>Entries:</p>
  <ul>
  {% for entry in entries %}
    <li>
    <p>
      {{ entry.date_added|date:'M d, Y H:i' }}
    </p>

    <p>
      {{ entry.text|linebreaks }}
    </p>
    </li>
  {% empty %}
    <li>There are no entries yet.</li>
  {% endfor %}
  </ul>

{% endblock content %}
```

## The Django shell

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

### Start a shell session

```
$ python manage.py shell
```

### Access data from the project

```
>>> from learning_logs.models import Topic
>>> Topic.objects.all()
[<Topic: Chess>, <Topic: Rock Climbing>]
>>> topic = Topic.objects.get(id=1)
>>> topic.text
'Chess'
>>> topic.entry_set.all()
<QuerySet [<Entry: In the opening phase...>]>
```

*More cheat sheets available at*

ehmatthes.github.io/pcc_2e/

# Beginner's Python Cheat Sheet - Django, Part 2

## Users and forms

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

## User accounts

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

### Making a users app

*After making the app, be sure to add `'users'` to `INSTALLED_APPS` in the project's settings.py file.*

```
$ python manage.py startapp users
```

### Including URLs for the users app

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

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('users/', include('users.urls')),
    path('', include('learning_logs.urls')),
]
```

## Using forms in Django

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

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

## User accounts (cont.)

### 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.*

```
from django.urls import path, include

from . import views

app_name = 'users'
urlpatterns = [
    # Include default auth urls.
    path('', include(
            'django.contrib.auth.urls')),

    # Registration page.
    path('register/', views.register,
            name='register'),
]
```

### The login template

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

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

```
{% extends "learning_logs/base.html" %}

{% block content %}

  {% if form.errors %}
  <p>
    Your username and password didn't match.
    Please try again.
  </p>
  {% endif %}

  <form method="post"
      action="{% url 'users:login' %}">

    {% csrf token %}
    {{ form.as_p }}
    <button name="submit">Log in</button>

    <input type="hidden" name="next"
      value="{% url 'learning_logs:index' %}"/>

  </form>

{% endblock content %}
```

## User accounts (cont.)

### Showing the current login status

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

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

```
<p>
  <a href="{% url 'learning_logs:index' %}">
    Learning Log
  </a>

  {% if user.is_authenticated %}
    Hello, {{ user.username }}.
    <a href="{% url 'users:logout' %}">
      Log out
    </a>
  {% else %}
    <a href="{% url 'users:register' %}">
      Register
    </a> -
    <a href="{% url 'users:login' %}">
      Log in
    </a>
  {% endif %}

</p>

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

### The logged_out template

*The default logout view renders the page using the template logged_out.html, which needs to be saved in the users/templates/registration/ folder.*

```
{% extends "learning_logs/base.html" %}

{% block content %}
  <p>
    You have been logged out. Thank you
    for visiting!
  </p>
{% endblock content %}
```

## User accounts (cont.)

### The register view

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

```
from django.shortcuts import render, redirect
from django.contrib.auth import login
from django.contrib.auth.forms import \
        UserCreationForm

def register(request):
    """Register a new user."""

    if request.method != 'POST':
        # Display blank registration form.
        form = UserCreationForm()

    else:
        # Process completed form.
        form = UserCreationForm(
                data=request.POST)

        if form.is_valid():
            new_user = form.save()

            # Log in, redirect to home page.
            login(request, new_user)
            return redirect(
                    'learning_logs:index')

    # Display a blank or invalid form.
    context = {'form': form}

    return render(request,
        'registration/register.html', context)
```

## Styling your project

*The `django-bootstrap4` app allows you to use the Bootstrap library to make your project look visually appealing. The app provides tags that you can use in your templates to style individual elements on a page. Learn more at django-bootstrap4.readthedocs.io/.*

## Deploying your project

*Heroku lets you push your project to a live server, making it available to anyone with an internet connection. Heroku offers a free service level, which lets you learn the deployment process without any commitment.*

*You'll need to install a set of Heroku command line tools, and use Git to track the state of your project. See devcenter.heroku.com/, and click on the Python link.*

## User accounts (cont.)

### The register template

*The register.html template shown here displays the registration form in paragraph format.*

```
{% extends 'learning_logs/base.html' %}

{% block content %}

  <form method='post'
      action="{% url 'users:register' %}">

    {% csrf_token %}
    {{ form.as_p }}

    <button name='submit'>Register</button>
    <input type='hidden' name='next'
      value="{% url 'learning_logs:index' %}"/>

  </form>

{% endblock content %}
```

## Connecting data to users

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

### Making a topic belong to a user

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

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

```
from django.db import models
from django.contrib.auth.models import User

class Topic(models.Model):
    """A topic the user is learning about."""
    text = models.CharField(max_length=200)
    date_added = models.DateTimeField(
            auto_now_add=True)

    owner = models.ForeignKey(User,
            on_delete=models.CASCADE)

    def __str__(self):
        return self.text
```

### Querying data for the current user

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

```
topics = Topic.objects.filter(
        owner=request.user)
```

## Connecting data to users (cont.)

### Restricting access to logged-in users

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

```
from django.contrib.auth.decorators import \
        login_required
--snip--

@login_required
def topic(request, topic_id):
    """Show a topic and all its entries."""
```

### Setting the redirect URL

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

```
LOGIN_URL = 'users:login'
```

### Preventing inadvertent access

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

```
from django.http import Http404
--snip--

@login_required
def topic(request, topic_id):
    """Show a topic and all its entries."""
    topic = Topics.objects.get(id=topic_id)
    if topic.owner != request.user:
        raise Http404
    --snip--
```

## Using a form to edit data

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

### Creating a form with initial data

*The `instance` parameter allows you to specify initial data for a form.*

```
form = EntryForm(instance=entry)
```

### Modifying data before saving

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

```
new_topic = form.save(commit=False)
new_topic.owner = request.user
new_topic.save()
```

*More cheat sheets available at*

ehmatthes.github.io/pcc_2e/

# Beginner's Python Cheat Sheet - Git

## Version Control

Version control software allows you to take snapshots of a project whenever it's in a working state. If your project stops working, you can roll back to the most recent working version of the project.

Version control is important because it frees you to try new ideas with your code, without worrying that you'll break your overall project. A distributed version control system like Git is also really useful in working collaboratively with other developers.

## Installing Git

*You can find an installer for your system at git-scm.com/. Before doing that, check if Git is already on your system:*

```
$ git --version
git version 2.20.1 (Apple Git-117)
```

## Configuring Git

*You can configure Git so some of its features are easier to use. The editor setting controls which editor Git will open when it needs you to enter text.*

### See all global settings

```
$ git config --list
```

### Set username

```
$ git config --global user.name "eric"
```

### Set email

```
$ git config --global user.email
    "eric@example.com"
```

### Set editor

```
$ git config --global core.editor "nano"
```

## Ignoring files

*To ignore files make a file called ".gitignore", with a leading dot and no extension. Then list the directories and files you want to ignore.*

### Ignore directories

```
__pycache__/
my_venv/
```

## Ignoring files (cont.)

### Ignore specific files

```
.DS_Store
secret_key.txt
```

### Ignore files with specific extensions

```
*.pyc
```

## Initializing a repository

*All the files Git uses to manage the repository are located in the hidden directory .git. Don't delete that directory, or you'll lose your project's history.*

### Initialize a repository

```
$ git init
Initialized empty Git repository in
    my_project/.git/
```

## Checking the status

*It's important to check the status of your project often, even before the first commit. This will tell you which files Git is planning to track.*

### Check status

```
$ git status
On branch main
No commits yet
Untracked files:
    .gitignore
    hello.py
    ...
```

## Adding files

*You'll need to add the files you want Git to keep track of.*

### Add all files not in .gitignore

```
$ git add .
```

### Add a single file

```
$ git add hello.py
```

## Making a commit

*When making a commit, the -am flag commits all files that have been added, and records a commit message. (It's a good idea to check the status before making each commit.)*

### Make a commit with a message

```
$ git commit -am "Started project, everything
    works."
2 files changed, 8 insertions(+)
create mode 100644 .gitignore
```

## Checking the log

*Git logs all the commits you've made. Checking the log is helpful for understanding the history of your project.*

### Check log in default format

```
$ git log
commit 7c0a5d8... (HEAD -> main)
Author: Eric Matthes <eric@example.com>
Date:   Mon Feb 15 08:40:21 2021 -0900
    Greets user.
commit b9aedbb...
...
```

### Check log in simpler format

```
$ git log --oneline
7c0a5d8 (HEAD -> main) Greets user.
b9aedbb Started project, everything works.
```

## Exploring history

*You can explore a project's history by visiting specific commit hashes, or by referencing the project's HEAD. HEAD refers to the most recent commit of the current branch.*

### Visit a specific commit

```
$ git checkout b9aedbb
```

### Return to most recent commit of main branch

```
$ git checkout main
```

### Visit the previous commit

```
$ git checkout HEAD^
```

### Visit an earlier commit

```
$ git checkout HEAD^^^
```

### Visit the previous commit

```
$ git checkout HEAD~1
```

### Vist an earlier commit

```
$ git checkout HEAD~3
```

## Learning more

*You can learn more about using Git with the command git help. You can also,go to Stack Overflow and search for git, and then sort the questions by number of votes.*

## Branching

*When the work you're about to do will involve multiple commits, you can create a branch where you'll do this work. The changes you make will be kept away from your main branch until you choose to merge them. It's common to delete a branch after merging back to the main branch.*

*Branches can also be used to maintain independent releases of a project.*

### Make a new branch and switch to it

```
$ git checkout -b new_branch_name
Switched to a new branch 'new_branch_name'
```

### See all branches

```
$ git branch
* new_branch_name
main
```

### Switch to a different branch

```
$ git checkout main
Switched to branch 'main'
```

### Merge changes

```
$ git merge new_branch_name
Updating b9aedbb..5e5130a
Fast-forward
 hello.py | 5 +++++
 1 file changed, 5 insertions(+)
```

### Delete a branch

```
$ git branch -D new_branch_name
Deleted branch new_branch_name
    (was 5e5130a).
```

### Move last commit to new branch

```
$ git branch new_branch_name
$ git reset --hard HEAD~1
$ git checkout new_branch_name
```

## Undoing recent changes

*One of the main points of version control is to allow you to go back to any working state of your project and start over from there.*

### Get rid of all uncommited changes

```
$ git checkout .
```

### Get rid of all changes since a specific commit

```
$ git reset --hard b9aedbb
```

### Create new branch starting at a previous commit

```
$ git checkout -b branch_name b9aedbb
```

## Stashing changes

*If you want to save some changes without making a commit, you can stash your changes. This is useful when you want to revisit the most recent commit without making a new commit. You can stash as many sets of changes as you need.*

### Stash changes since last commit

```
$ git stash
Saved working directory and index state
    WIP on main: f6f39a6...
```

### See stashed changes

```
$ git stash list
stash@{0}: WIP on main: f6f39a6...
stash@{1}: WIP on main: f6f39a6...
...
```

### Reapply changes from most recent stash

```
$ git stash pop
```

### Reapply changes from a specific stash

```
$ git stash pop --index 1
```

### Clear all stashed changes

```
$ git stash clear
```

## Comparing commits

*It's often helpful to compare changes across different states of a project.*

### See all changes since last commit

```
$ git diff
```

### See changes in one file since last commit

```
$ git diff hello.py
```

### See changes since a specific commit

```
$ git diff HEAD~2
$ git diff HEAD^^
$ git diff fab2cdd
```

### See changes between two commits

```
$ git diff fab2cdd 7c0a5d8
```

### See changes in one file between two commits

```
$ git diff fab2cdd 7c0a5d8 hello.py
```

## Good commit habits

*Try to make a commit whenever your project is in a new working state. Make sure you're writing concise commit messages that focus on what changes have been implemented. If you're starting work on a new feature or bugfix, consider making a new branch.*

## Git & GitHub

*GitHub is a platform for sharing code, and working collaboratively on code. You can clone any public project on GitHub. When you have an account, you can upload your own projects, and make them public or private.*

### Clone an existing repository to your local system

```
$ git clone
    https://github.com/ehmatthes/pcc_2e.git/
Cloning into 'pcc_2e'...
...
Resolving deltas: 100% (816/816), done.
```

### Push a local project to a GitHub repository

*You'll need to make an empty repository on GitHub first.*

```
$ git remote add origin
  https://github.com/username/hello_repo.git
$ git push -u origin main
Enumerating objects: 10, done.
...
To https://github.com/username/hello_repo.git
 * [new branch]      main -> main
Branch 'main' set up to track remote branch
    'main' from 'origin'.
```

### Push recent changes to your GitHub repository

```
$ git push origin branch_name
```

## Using pull requests

*When you want to pull a set of changes from one branch into the main branch of a project on GitHub, you can make a pull request. To practice making pull requests on your own repositories, make a new branch for your work. When you're finished the work, push the branch to your repository. Then go to the "Pull requests" tab on GitHub, and click "Compare & pull request" on the branch you wish to merge. When you're ready, click "Merge pull request".*

*You can then pull these changes back into your local main branch with `git pull origin main`. This is an alternative to merging changes to your main branch locally, and then pushing the main branch to GitHub.*

## Practicing with Git

*Git can be used in simple ways as a solo developer, and complex ways as part of a large collaborative team. You can gain valuable experience by making a simple throwaway project and trying all of these steps with that project. Make sure your project has multiple files and nested folders to get a clear sense of how Git works.*

*More cheat sheets available at*

ehmatthes.github.io/pcc_2e/