Python for Data Analysis

Lecture Notes

Alexander Clark

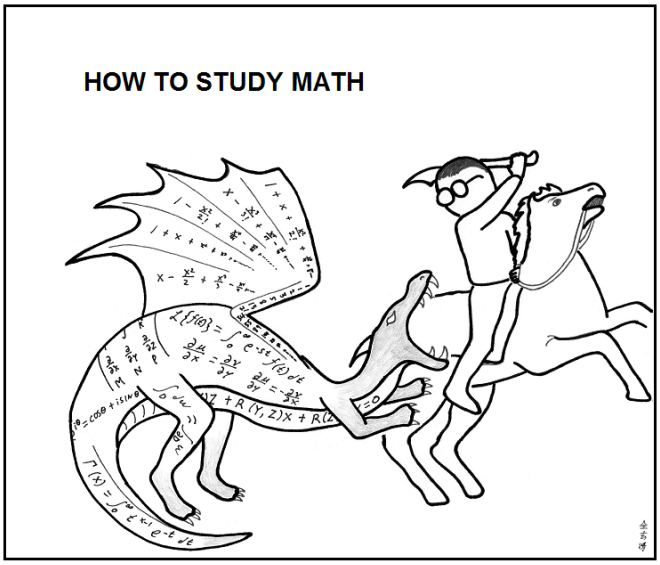
Columbia University SPS

To the reader: These are my lecture notes for Python for Data Analysis, which I consider a work in progress. They're not written to replace class attendance, so you might not find them self-contained. These notes rely on three books,

- *Starting out with Python* by Tony Gaddis
- *Python for Data Analysis* by Wes McKinney
- *Python Data Science Handbook* by Jake VanderPlas.

There are many other fantastic books and resources for self-studying Python. I encourage you to find as many supplementary sources as you find helpful, but to return to these notes to help focus on what will be important for doing well on assignments and exams in this course.

Famous mathematician Paul Halmos counseled students not to study math too passively. With Python, this is also good advice. Ease into the language, but don't become too satisfied with running others' code or making only minimal edits. Work from scratch and relish the detours.



Don't just read it; fight it!

--- Paul R. Halmos

Source: AbstruseGoose.com

Contents

Part I

Lecture 1

1 Course Overview

Welcome! Please check Canvas for important announcements, materials, and links throughout the semester. You'll also find course materials on Google Drive. Send me a request for access if the folder is not already shared with you.

The point of the class is to learn Python *for* data analysis. This is not a computer science class and we will avoid taking too many detours that do not add meaningfully to a data analyst's/scientist's skillset. That means we'll learn enough syntax to manipulate data, analyze data, comfortably use existing packages, and write code for simple enough tasks where there might not be a suitable function we can pull of the shelf.

We'll also be using Python 3. Syntax for Python 2 is slightly different, so don't be surprised if you find some discrepancies in syntax in old Stack Exchange posts and in other resources.

1.1 Faculty

```
instructor, instructor_email = "Alexander Clark", "ac4725@columbia.edu"
# associate, associate_email =
```

We're available by appointment and we'll set office hours based on the week.

1.2 Books & My Programming Lab

To start, we're working from *Starting out with Python* by Tony Gaddis. We'll use the accompanying My Programming Lab (MPL) exercises. I'll have more to say about MPL later. My tentative plan is that MPL will be graded only to the requirement that you try half of the exercises.

1.3 Software

I think it's important we aren't dogmatic about the IDE or software we use, but you must download Anaconda. Contact me or the associate if you have trouble with the installation. We'll work with Jupyter and Spyder. I also recommend you download Sublime Text or another simple text editor with synatax highlighting. I will also use Google Colaboratory until we formally introduce Jupyter later in the course.

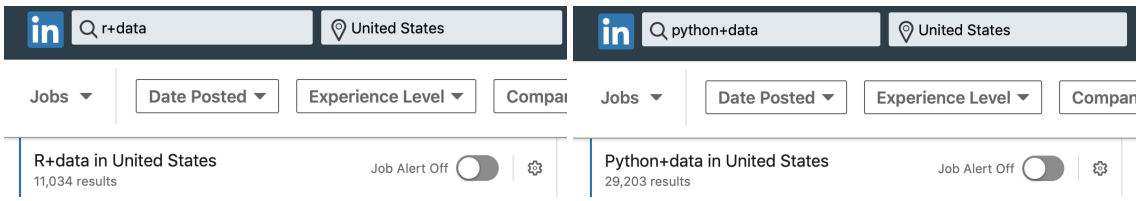
2 Why Python?

First, what is Python? It's a high-level, general-purpose programming language. It is more general-purpose than a language like R. With libraries like pandas and scikit-learn available, it is now popular for data analysis.

Python's advantages over R are its readability and its popularity.

See the Zen of Python for design guidelines and Pep 8 for a coding style guide.

You'll see Python in more job listings. I use Python every day, and I use R only occasionally for its more advanced statistical packages.



🎤💉.

3 Input, Processing, and Output

Reference: Gaddis Chapter 2

3.1 The Hello World Program (G§2.3)

To get started in any langauge, printing "Hello, World!" might be the first step.[1]

In Python, we can print an input using the `print()` function. We simply pass our desired input within the parentheses, and Python will print the value.

We can enter text as a **string**. Text entered inside single, double, or triple quotations is interpreted as a string.

[1] See the Wikipedia page for Hello, World.

```
print('Hello, World')
print("Hello, World!")
print("""Hello,
World!""")
```

In Python 2, the syntax would have been `print 'Hello, World!'` without the parentheses.

3.2 Comments (G§2.4)

Commenting your code is helpful if you care about your colleagues or your future self. Comments should add clarity to the intention and workings of code. A comment is a piece of code that isn't actually executed—it's a comment left for the reader or the person who inherits and modifies your code. Everything after a `#` will be ignored by the Python interpreter.

```
# This will print a greeting.
print('Hello, World!')
```

You might also use end-line comments like the following

```
print('Hello, World!') # Prints a greeting
```

PEP 8 addresses comments here. I don't intend to grade based on the stylistic orthodoxy of your comments, but spaces are free so I do recommend `# Comments like this` instead of `#Comments like this`.

3.3 Variables (G§2.5)

A **variable** holds a value. It can be a string, a number, or perhaps a more complicated data type. Variable assignment is done with the equals sign, `=`.

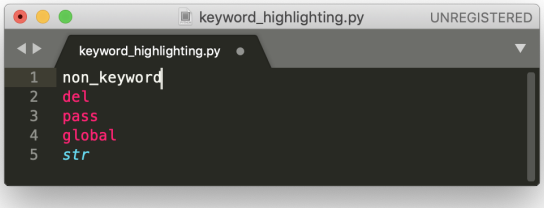
```
greeting = "Hello, World!"
my_favorite_number = 91
```

Now compare the output you get from the following.

```
print(greeting)
print('greeting')
print("Hello, World!")
print(91)
print(my_favorite_number)
print('my_favorite_number')
print("My favorite number is", my_favorite_number)
print("My favorite number is ", my_favorite_number)
```

PEP 8 addresses variable names here. Use lowercase and underscores. This is good advice, but I don't see any reason to be too wedded to this. If you want to assign a matrix to a variable, it's reasonable to use an uppercase letter as the variable name. There, a math convention overrides a Python convention. Indeed, Gaddis is fine with uppercase (p. 43).

More importantly, avoid Python key words in your variable names. Here is a list of keywords which have a specific meaning in code. An IDE will make this easier for you by highlighting keywords.



3.4 Data Types and Conversion (G§2.5, 2.6)

To start, we are concerned with strings, integers, and floats. In Python, these are classes `str`, `int`, and `float`. You can check the type of variable or value using `type()`.

```
string_example = ''
int_example = -1
float_example = -1.
```

Some types can be converted by using `str()`, `int()`, or `float()`.

3.5 Input (G§2.6)

It's not that common for a data science workflow, but you can read input using `input()`. The input is always read in as a string.

```
favorte_color = input("What is your favorite color?")
favorite_number = input("What is your favorite number?")
attending_in_person = input("I am attending class in person.")
```

3.6 Calculations (G§2.7)

You can use Python as a calculator. See the list of operations and symbols in Table 2-3 (Gaddis p. 54). What might stand out is

- Exponentiation is done with `**`, not `^`.
- Integer division (rounds down) is done with `//`.
- The remainder of x divided by y can be found with `x % y`, which might be read as x *modulo* y.

If a float is involved in an operation, the result will also be a float.

4 Decision Structures

Reference: Gaddis Chapter 3

Decision structures allow a program to have more than one path of execution. The path depends on condition. The condition is either True or False, and so can be represented by a Boolean variable.

4.1 If Statements (G§3.1, 3.3)

Here's a joke. A programmer is going to the grocery store and his partner says, "Buy a gallon of milk, and if there are eggs, buy a dozen." The programmer comes home with 13 gallons of milk.

Or consider the logical inference if you ask, "Is it raining?" and get a reply, "Not hard."

```
if 2 + 2 > 4:
    print("Pigs can fly.")

if 2 + 2 == 4:
    print("Pigs cannot fly.")

if 'a' < 'b':
    print("It is true that 'a' is less than 'b'.")

if 'a' < 'A':
    print("It is true that 'a' is less than 'A'.")

if 'goon' == 'Goblin':
    print("A goon is a goblin.")
```

If statements like the above rely on *relational operators* (see Table 3-1 Gaddis p.112).

```
if 2 == 2.:
    print("The integer and the float are equal.")

if 2 is 2.:
    print("The integer and the float are the same object in memory.")
```

4.2 If Elif Else Statements (G§3.2, 3.4)

Compare the output from the following programs.

```
num = 0

if num < 1:
    print(num)
    num = num + 2
if num > 0:
    print(num, '!')
    num = num - 1000
if True:
    print(num, '?')
```

```
num = 0

if num < 1:
    print(num)
    num = num + 2
elif num > 0:
    print(num, '!')
    num = num - 1000
else:
    print(num, '?')
```

4.3 Logical Operators (G§3.5)

Suppose you want to execute some code if a number x is between 10 and 20. You could use *nested* if statements.

```
x = 14
if x >= 10:
    if x <= 20:
        print("x is between 10 and 20.")
```

But you might prefer to base your if statement off of one compound Boolean expression. For these, we need logical operations. They are

- Logical *and*: `and`
- Logical *or*: `or`
- Logical *negation*: `not`

Observe the following will give equivalent output.

```
not 1 > 2
not (1 > 2)
not(1 > 2)
```

Challenge: What do you expect from `not not (1 or False)`?

4.4 Boolean Variables (G§3.6)

Finally, a Boolean variable simply references a logical True or False.

```
# Option Value
market_value = 10
strike_price = 9
option_has_value = market_value > strike_price

if option_has_value:
    print("We're in the money.")

# Check data type
print(type(option_has_value)
print(type(False)))
```

Part II

Lecture 2

We now touch on lists (from Chapter 7) and then go back to Chapters 4 and 5 to cover repetition structures and functions.

5 Lists

Reference: Gaddis Chapter 7

5.1 Defining Lists (G§7.2)

We have worked with strings, integers, floats, and booleans so far. Now, we introduce a compound data type the list. Lists are also a sequence object, meaning they are ordered.

A list is constructed by separating one or more objects with commas and placing them in square brackets. For example, we can define a list as follows.

```
# Our first list---some Peloton content categories
floor = ['yoga', 'meditation', 'stretching', 'strength', 'cardiovascular']
```

```
# More lists
aquatic = []
cycling = ['cycling']
```

```
# More lists
treadmill_based = ['running', 'walking', 'bootcamp']
```

A useful quality of lists is that they can be of mixed type.

```
# More lists
fine_list = [0, 'milk', cycling]
```

5.2 Indexing, Slicing, Mutating (G§7.2, 7.3)

You can obtain the element at a certain place, or *index*, in a list by suffixing the list with that index number inside square brackets. Python uses zero-based indexing. So one might say the n^{th} item is at index $n-1$. This can create confusion when talking about the first, second, etc items in a list. I'll try always to speak of elements *at index* zero, one, etc, so that you know I am referring to the position according to this zero-based system.

```
# Get specific items
fine_list = [0, 'milk', cycling]

# Which of these print statements will raise an error?
print(fine_list[0])
print(fine_list[1])
print(fine_list[2])
print(fine_list[3])
print(fine_list[-1]))
```

We can also count from the end of the list to find an item, as hinted by the `fine_list[-1]` above. The index -1 will identify the last element. So, in some sense, you might say that negative indexing is not zero-based though it is in fact consistent with the zero-based indexing described above. If that's confusing, recall $-0 = 0$ and so `fine_list[-0]` is actually the same as `fine_list[0]`.

The following graphic helps explain how lists (and strings) are indexed.

While indexing pulled out a single element, we can also *slice* to pull out a sublist.

```
# Get specific items
['Jack', 'Jill', 'and', 'hill', 'over', 'ran', 'the']
print(fine_list[0:2])
print(fine_list[-2:])
```

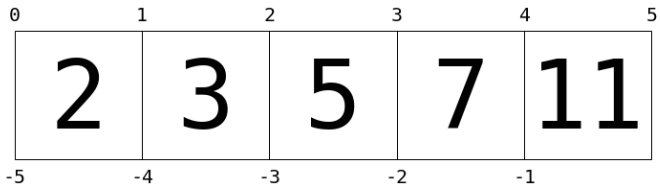


Figure 1: Indexing (From *Whirlwind Tour of Python* by Jake VanderPlas).

The first print statement will print elements 0 and 1 in a list. The last print statement will print a list of just the last two elements.

Lists are *mutable*, meaning that we can change their contents.

```
# Get specific items
fine_list = [0, 'milk', cycling]
print(fine_list[0])
fine_list[0] = 1.0
print(fine_list[0])
```

6 Repetition Structures

Reference: Gaddis Chapter 4

Repetition structures (loops) are one of the best justifications for moving from Excel to Python (though they are not unique to Python). A "program" in Excel is a serious of keystrokes and clicks. You might create a report for your company that is specific to one market and you'll need to replicate the same report for a different market. Perhaps you could write a macro, but I think you'll find working in Python to be easier. In Python, we can do this in a loop. We can have a program that makes the report and we can iterate through the different markets to apply the program to each and create the specific reports (using a for loop).

6.1 While Loops (G§4.2)

The **while loop** is a condition-controlled loop. Figure 2 illustrates the logic well. See also Figure 2 in Gaddis §4.2 (p. 163).

The statement inside a while loop executes as long as the condition evaluates to True.

```
# Rest on the seventh day
day_of_week = 1
while day_of_week < 7:
    print('work')
    day_of_week += 1
```

Beware the infinite loop. Be confident that your test condition will have a way of becoming False, otherwise you might notice that your program never finishes.

Let's consider an infinite series, $\sum_{i=0}^{\infty} 2^{-i} = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots = 2$. We could try to calculate this with a while loop if we didn't know the sum converged to two.

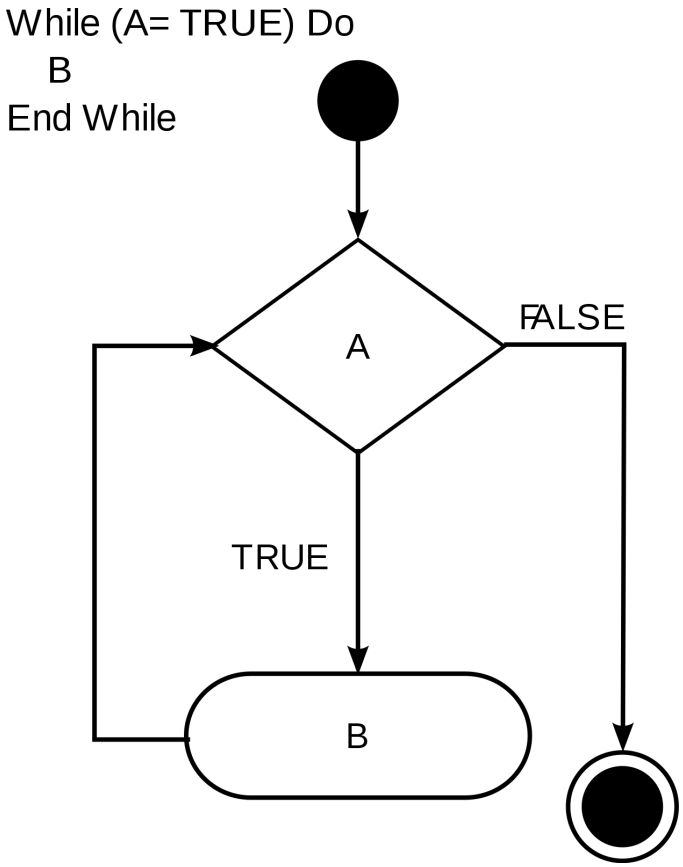


Figure 2: While loop logic (Wikipedia).

```
# Sum of a geometric series
the_sum = 0
idx = 0
increment = 2 ** -idx

while increment > 0:
    # Increase the sum by the current increment
    the_sum += increment

    # Advance the index in the sum and calculate a new increment
    idx += 1
    increment = 2 ** (-idx)
```

Does this loop make you nervous? In fact, the loop will terminate because we will eventually hit machine zero. I found the loop to terminate at the increment 2^{-1075} and the resulting sum to be two. But it should make you nervous. You might instead decide on some level of precision and use a test condition like `increment > .0005`, in which case you could find a bound on the error with some math. Doing that math is not part of this course, (un)fortunately.

6.2 For Loops (G§4.3)

For loops execute the attached code based on some iteration. The code might depend on a variable that is actually changing with the iteration.

```
# Dr. Seuss
for item in [1,2,'red','blue']:
    print(item, 'fish')
```

The for clause tells Python to execute the statement once for every item in the iterable, which is in this case the object `[1,2,'red','blue']`. This is a *list*, a kind of compound data object that can store other objects in sequence by separating them with commas and putting them inside square brackets. Below we use `range(5)` instead of a list, which you can think of as generating an iterable object of integers from 0 to 4 (5 is not included, but the length is 5).

Or you might just want to execute a specific set of statements some number of times.

```
# Tubthumping by Chumbawamba
for item in range(5):
    print("I get knocked down, but I get up again.")
    print("You are never gonna keep me down.")
```

For both of the examples above, `item` is the *variable*. However, notice the print statement only depends on the variable in the first example.

You can even iterate over the characters in a string.

```
# Cheer
word = 'PYTHON'
for char in word:
    print("Give me a",char, '!')
    print("    ", char)
print("What's that spell?")
print("    ",word)
```

6.3 Nested Loops (G§4.7)

A loop that is inside another is called *nested*.

Think about how Python executes code line by line. What order of output do you expect from this program?

```
# Nested Loop
for x in ['wee', 'bee']:
    for y in ['bop', 'dop']:
        print(x,y)
```

7 Functions

Reference: Gaddis Chapter 5

Functions make your code easier to read and reuse by performing a certain task based on some number of arguments the function takes.

In math, you might define a function $f(x) = x^2$. Just as f does something with x, a function like `print` does something with whatever input we pass inside the parentheses.

Consider the cheer we made in Section 6.2.

```
# Cheer
word = 'PYTHON'
for char in word:
    print("Give me a",char, '!')
    print("    ", char)
print("What's that spell?")
print("    ",word)
```

We can make this into a function.

```
# Cheer Function
def cheer(word):
    ''' Doc string '''
    for char in word:
        print("Give me a",char, '!')
        print("    ", char)
    print("What's that spell?")
    print("    ",word)
```

Here's another function that takes a number and returns the next multiple of five by making use of a while loop. This is our first *value-returning* function. Don't confuse the idea of a function returning something with simply doing something. The `cheer` function merely did something by printing a cheer.

```
# Find a nearby multiple of five
def find_close_multiple_of_five(num):
    # check if multiple of five
    is_multiple = num % 5 == 0
    while is_multiple not True:
        num += 1
        is_multiple = num % 5
    return num
```

8 Exercise

Hard: Write a program that will find the first prime number after some given integer.

Answers on next page.

8.1 Exercise Answers

Hard: Write a program that will find the first prime number after some given integer.

```
# Find the first prime number greater than num.
# Initialize with a composite number
num = 20
is_prime = False

# Incremement the number until we find a prime.
while is_prime == False:
    num += 1
    is_prime = True # Initialize as prime and overwrite later

    for divisor in range(2,num): # Checks all integers >=2 and < num
        if num % divisor == 0: # We enter the if statement when num/divisor is a whole number
            is_prime = False and is_prime # False along with and will make the whole boolean False
    # is_prime remains True when we exit the for loop not having found a single divisor
```

Part III

Lecture 3

9 Functions

Reference: Gaddis Chapter 5

The motivation for functions is to create more readable and modular code. Function definitions take a basic form like the following.

```
# Minimal Example (void function)
def say_hi():
    print("hello")
```

We use the keyword `def`, followed by the function name, parentheses, and a colon. Then, in an indented block, you write what you want the function to do. More elaborate functions will involve *arguments* and *return statements*. Specifying an argument allows for the code executed by the function to depend on one or more variables.

```
# Minimal Example with An Argument
def say_hi(name):
    print("hello", name)
```

Sometimes we want to get some object and possibly assign it to a variable. In this case, you should use a `return` statement.

```
# Minimal Example (value-returning function)
def my_favorite_number():
    return 91
```

And let's combine them.

```
# Minimal Example with a Return and Arguments
def add_numbers(x,y):
    return x + y
```

9.1 Local Variables (G§5.4)

The variables created inside a function are called *local variables*. They are local in the sense that they are accessible only within the operation of the function.

```
my_favorite_number = 91 # global variable

def opposite_day():
    my_favorite_number = -91 # local variable
    return my_favorite_number

print(opposite_day())
print(my_favorite_number)
```

The *scope* of a variable is the part of a program in which the variable can be accessed. A local variable's scope is the function in which the variable is created.

9.2 Global Variables (G§5.6)

A *global variable* is accessible to all the functions in a program file.

Contrast

```
number = 0

def identity(number):
    return number

print(identity(900))
print(number)
```

and

```
# This will return an error
number = 0

def identity(number):

    return number

print(identity(900))
print(number)
```

9.3 Naming Conventions and Other Best Practices (PEP 8)

As discussed in Gaddis §5.2, a function should be given a descriptive name, so that a reader can guess what the function might do. The same considerations and rules when naming variables apply here (see PEP 8).

9.4 You Can Put Functions in Lists

Operators like + or > can't go in a list. However, functions can.

```
# This will give an error.
for operator in [+,>]:
    print(1 operator 2)
```

Try this instead.

```
def add(x,y):
    return x + y
def greater_than(x,y):
    return x > y

for function in [add, greater_than]:
    print(function(1,2))
```

10 Exceptions

Reference: Gaddis Chapter 6

Occasionally, you might ask Python to do something impossible. Try running `1/0`. You will get an error message, `ZeroDivisionError`. Robust code should deal with this possibility intelligently. This is especially true when writing functions. You can avoid errors by writing your code to prevent them from occurring. Or, you might write code that responds to errors. For more on the built-in exceptions, follow this link.

Here's an example of avoiding the division by zero error.

```
def pct_change(old, new):
    delta = new - old
    if old != 0:
        pct = 100 * delta / old
        return pct
    else:
        return "Not defined."
```

Here's an example of dealing with the error, which requires `try` and `except` statements.

```
def pct_change1(old, new):
    delta = new - old
    try:
        pct = 100 * delta / old
        return pct
    except ZeroDivisionError:
        return "Not defined."
```

```
# Will this work?
def pct_change2(old, new):
    delta = new - old
    try:
        pct = 100 * delta / old
        return pct
    return "Not defined."
```

A `try` must be paired with an `except`, so the definition of `pct_change2` will not work. There must be an `except` and, as we use in `pct_change1`, it's a good idea to use the form `except ExceptionName`. This tells Python what code to run when a certain exception is encountered.

```
def pct_change3(old, new):
    try:
        delta = new - old
    except TypeError:
        return "Use ints or floats."
    try:
        pct = 100 * delta / old
        return pct
    except ZeroDivisionError:
        return "Not defined."
```

The following function, `pct_change4`, actually won't run properly if you attempt to execute `pct_change4(1,'cheese')`. Can you figure out why? Think about the ordering of the code.

```
def pct_change4(old, new):
    delta = new - old
    try:
        pct = 100 * delta / old
        return pct
    except ZeroDivisionError:
        return "Not defined."
    except TypeError:
        return "Use ints or floats."
```

You do not need to specify the error type in your `except` statement. Sometimes this might hide errors that you do want to be surfaced or stop the execution of a program, so use these blanket exceptions carefully.

```
def pct_change5(old, new):
    try:
        delta = new - old
        pct = 100 * delta / old
    except:
        return "An error occurred."
    return pct
```

You might use a blanket except after handling specific errors.

```
def pct_change6(old, new):
    try:
        delta = new - old
        pct = 100 * delta / old
        return pct
    except ZeroDivisionError:
        return "You can't divide by zero."
    except:
        return "An error occurred."
```

It can be useful information to know what type of exception your code generates. In that case, you can access and print that error.

```
try:
    'a' + 1
```

```
except Exception as e:
    print(e)
```

The use of `Exception` above matters. Compare these two programs.

```
try:
    'a' + 1
except TypeError as e:
    print(e)
```

```
try:
    'a' + 1
except ValueError as e:
    print(e)
```

Only the first program above actually handles the exception.

Now, we consider the use of `else` and `finally` statements after a `try`/`except`. An `else` clause can be added after the `except` clauses, and the statements in the `else` clause are executed only if no exceptions were raised. The *else* is then in contrast to the raising of exceptions, in a similar style as when an `else` might be used in complement to an `if`. What output do you expect from the following program? What if we changed the value of `denom`?

```
denom = 0
try:
    print(2 / denom)
except Exception as e:
    print(e)
else:
    print(3 / denom)
```

Here are slightly more complicated examples. Try running them to see what happens.

```
products = ['bike','treadmill']
try:
    print(products[2])
except IndexError as e:
    print(e)
except Exception as e:
    print(e, 'other error')
else:
    print(products[2], '!')
```

```
products = ['bike','treadmill']
try:
    print(products[1])
except IndexError as e:
    print(e)
except Exception as e:
    print(e, 'other error')
else:
    print("Else block time")
    print(products[2], '!')
```

Next, we introduce the `finally` clause. Whereas the `else` block was executed when no exceptions were raised, the `finally` closed is executed no matter what. This StackExchange post helps explain the unique usefulness of this, but we won't go into that much detail.

```
products = ['bike','treadmill']
idx = 2
try:
    print(products[idx])
except Exception as e:
    print(e, 'other error')
finally:
    print('End')
```

11 Tuples

Finally, a `tuple` is a lot like a list. Whereas lists used square brackets (`[]`), a tuple uses parentheses, (`()`). Like lists, we can index and slice a tuple. The main difference is that tuples are immutable. We cannot reassign the element at a particular index.

```
example_list = ['bike','treadmill']
example_tuple = ('bike','treadmill')

example_list[1] = 'treadmill'
example_tuple[1] = 'spacecraft' # This will throw an error
```

Finally, note we can convert lists and tuples, similar to the way we could convert floats and strings (`int("1")`, `str(1)`).

```
example_list = ['bike','treadmill']
example_tuple = ('bike','treadmill')

print(list(example_tuple))
print(tuple(example_list))
```

Part IV

Lecture 4

12 Tuples Revisited

Reference: Gaddis Chapter 7.9

Recall the primary difference between lists and tuples: lists are mutable and tuples are not. It might be hard to see immutability as an advantage, but this does make tuples to be safer objects. You won't accidentally screw them up (unless at the very beginning).

There is one definite advantage to tuples. They are processed faster. Processing speed will not be a practical concern in this class, but it could be a concern in your professional career.

13 Lists Revisited

Reference: Gaddis Chapter 7.4 and 7.5

13.1 The `in` Operator (G§7.4)

The `in` operator allows you to determine if an element is contained in a list. In Python, the statement `x in A`, where `A` is a list, mirrors the mathematical statement $x \in A$ where A is a set. While we never formally introduced `in`, we've already seen it in for loops with the for clause, `for item in range(10):`, for example.

Realize that for numerics, `in` will evaluate to true as long as the element is equal (==) to something in the list. That means a float can be `in` a list of integers.

```
chapters_on_midterm = [2,3,4,4,5,6,7,8,9]

# confirm all integers
for item in chapters_on_midterm:
    print(item, type(item))

# This is certainly True
bool1 = 2 in chapters_on_midterm

# What about this?
bool2 = 2.0 in chapters_on_midterm # This is True!
```

Finally, you can use `not in` as you might expect. The statement `x not in some_list` if there is no element `y in some_list` that is equal to `x`.

13.2 List Methods (G§7.5

Python methods are like functions, but they are associated with objects.

Functions look like this: `sorted(some_list)`
Methods look like this: `some_list.sort()`

For now, we can proceed thinking of them just as functions with this syntax.

The function `sorted()` and the method `.sort()` do the same thing in some sense—both can be used to sort a list. They are different in that `sorted()` returns a new sorted list without mutating `some_list`. The method `.sort()` doesn't return anything; it mutates the list into a sorted list. This difference isn't a property of functions and methods. The difference is particular to `sorted()` and `.sort()`.

```
# Demonstration of sorted() and .sort()

presorted_list = [1,2,3,4]
alt_list = [1,4,2,3]

c1 = alt_list == presorted_list
print(c1)

c2 = sorted(alt_list) == presorted_list
print(c2)

c3 = alt_list == presorted_list
print(c3)

alt_list.sort()

c4 = alt_list == presorted_list
```

```
print(c4)
```

Other important list methods include `.append()` and `.index()`. `.append()` requires an argument—an element to be added to the end of the list.

Running `some_list.append(x)` does the same thing `some_list += [x]` would accomplish.

```
# .append() Demonstration
ones = list()

ones.append(1)

print(ones)

ones += [1.0] # Recall this is the same as ones = ones + [1.0]

print(ones)
```

Next, the `.index()` method helps us find the index of the first instance of a particular element in a list. This method requires an argument, and `some_list.index(x)` returns the minimum index of `x` in `some_list`.

```
# .index() Demonstration
ones = [1, 1.0]

print(ones.index(1))

print(ones.index(1.0))

# There is no distinction between int and float
print(ones.index(1.0) == ones.index(1))
```

Exercise: Remove the duplicates from a list.

```
big_list = [1,2,4,2,12,4,12,234,1,1,1,1] # a lot duplicates

big_list_without_dups = []

for element in big_list:

  if element not in big_list_without_dups:

    big_list_without_dups.append(element)


## Alternate version

big_list_without_dups_alt = []

idx = 0 # keep track of index of each element
for element in big_list:
    # Add the element if it's the first time we've seen it in big_list
    first_index = big_list.index(element)
    if idx == first_index:
        big_list_without_dups_alt.append(element)

    # Advance the index for the next element
    idx += 1
```

14 Strings

Reference: Gaddis Chapter 8

Strings behave like lists in terms of indexing, slicing, and iterating. Strings are immutable however.

The `in` operator can be used on strings to test if one string is a substring within another string.

```
for char in 'team':
    print(char, char in team):

print("I" in 'team')
```

Important string methods include `.upper()`, `.lower()`, `.isalpha()`, `.split()`, and `.replace()`.

The `.upper()` and `.lower()` methods return a new string in uppercase or lowercase letters, respectively. These do not mutate the original string and no arguments are necessary.

The `.isalpha()` method returns a boolean, stating whether or not the string contains all alphabetical characters (this is different than not being a integer data type for example—a string might still contain a numeric character).

```
# In case you can't trust your eyes for l vs 1.

lowercase_L = "l"
one = "1"

for item in lowercase_L, one:
    print(item, item.isalpha())
```

The `.replace()` method takes two arguments. It returns a new string that replaces every instance of the first argument with the second argument. The function below returns a more muted string by eliminating all capital letters and converting exclamation marks to periods.

```
def lower_your_voice(string):
    lowercase = string.lower()
    not_exclamatory = lowercase.replace("!", ".")
    return not_exclamatory

print(lower_your_voice("Your card was declined!"))
```

The `.split()` method requires an argument and returns a list, dividing a string into substrings based on the argument. This is especially useful for splitting a sentence into its individual words.

```
invisible_hand = "It is not from the benevolence of the butcher that we expect our
    dinner but from their regard to their own interest"

the_words = invisible_hand.split(" ")

print(the_words)


# Note what happens when you split on the first or last character in a string

laugh = 'hahahahah'

split_laugh = laugh.split('h')
```

```
13
14 print(split_laugh)
15
16 # Note what happens when you pass no argument
17 print(laugh.split())
18
19 # This gives an error. You can't split on a zero-length separator.
20 laugh.split("")
```

15 Dictionaries

Reference: Gaddis Chapter 9

Dictionaries allow us to store key-value pairs. It's kind of like having one row in a table. Keys are like column names and values are the row-column cell value. Key-value pairs are created as `key: value`, then separated by commas and wrapped in curly braces, `{}`, to create a dictionary. Consider the example below.

```
1 workout = {'user': 'Velma', 'fitness_discipline': 'cycling', 'instructor': 'Matt
      Wilpers'}
```

A specific value is access by indexing the dictionary by the key.

```
1 print(workout['user'])
```

We can add new key-value pairs by assigning the value to the dictionary at that key.

```
1 # Build a dictionary from scratch
2 journal = dict() # creates an empty dictionary, can also use {}
3
4 journal['2020-10-03'] = "Today I learned a lot of Python. It was buckets of fun."
```

The `in` operator works on dictionaries by searching the keys.

```
1 # Help translate bad journalism
2 media_translator = {'is caused by': 'is correlated with'}
3
4 print('is caused by' in media_translator)
5 print('is correlated with' in media_translator)
```

You can access the keys with the `.keys()` operator and values with the `.values()` operator. So `x in some_dict` is actually a shorthand for `x in some_dict.keys()`.

Dictionary keys must be immutable. Tuples are fine.

```
1 # Economist Santa
2
3 gifts = {} # could also use dict() here
4
5 for child in ['Anna', 'Boris']:
6     for year in [2020, 2021]:
7
8         key = child, year # this is a tuple just like (child, year)
9
10        gifts[key] = 'money'
11
12 print(gifts)
```

16 Sets

Reference: Gaddis Chapter 9

I find sets to be underrated. They give flexibility in analysis because they allow for quick intersections and unions (e.g. find and analyze users who did this *and/or* that).

Sets are unordered and duplicates are ignored. We construct them like lists, but use `{}` instead of `[]`. Note that to create an empty set, you should use `set()` because `{}` creates an empty dictionary.

Being unordered means it is true that `{1,2} == {2,1}`.

It's not exactly right to say that sets *can't* have duplicates. You can create a set with duplicate elements and no error will be thrown. But those duplicates are ignored so that the created set object will not in fact have any duplicates. Thus, it is true that `{1,2} == {2,1,1,2}`.

Three important methods are `.union()`, `.intersection()`, and `.difference()`. Each of these acts on a set and requires another set as an argument. Intersection and union work like the set operations $\cap$ and $\cup$. The difference method performs set subtraction, $\setminus$. Recall that set subtraction is not commutative; $A \setminus B \neq B \setminus A$ unless $A = B$.

```
# Union and Intersection

primes = {2,3,5}
evens = {2,4,6}

even_and_prime = primes.intersection(evens)

even_or_prime = primes.union(evens)

for set_ in even_and_prime, even_or_prime: # note we're iterating over a tuple
    print(set_)

# Set Subtraction

contiguous_USA = {'New York', 'Kentucky', 'Wisconsin', 'California'} # among other
    states

tectonic_seceder = {'California'}


print(contiguous_USA.difference(tectonic_seceder))

# We can also use the - operator
print(contiguous_USA - tectonic_seceder)

## More Subtraction

cold_places = {'Wisconsin', 'Yukon'}

print(contiguous_USA.difference(cold_places))

# Reverse the arguments
print(cold_places.difference(contiguous_USA))
```

Part V

Lecture 5

17 Modules

17.1 Modules (Gaddis Appendix E)

A *module* is a file that contains Python code. Large programs are more manageable when divided into modules.

Many functions in the standard Python library are stored in modules. The `math` and `random` modules are common examples. These shouldn't require additional installation to use if you've downloaded Anaconda.

To use a module, you must import the module with an import statement, `import math` for example. Then any function in that module can be accessed by using the function name, prefixed by the module name and a dot. To use `sqrt` from `math`, you must use `math.sqrt(81)`.

You can import just a specific function with syntax like the following: `from math import sqrt`. Then `sqrt` can be accessed without the `math.` prefix. Running `from math import *` is called a wildcard import and will import every function in the module.

Finally, you can *alias* a specific module, library, or function with an `as` clause. There are conventional aliases for many modules. Pandas, NumPy, and Datetime are libraries we will cover later. These are typically imported with aliasing as in the below.

```
import pandas as pd
import numpy as np
import datetime as dt
from math import sin as sine # Not a typical alias, but for demonstration
```

17.2 Storing Functions in Modules (G§5.10)

You can create your own module by placing code into a .py file. If that file is in your directory, you can access it with an import statement. In the lecture folder, I've placed a file `next_power.py`. Download that and place it in your working directory to try importing it. In Google Colab, you can upload files in the left menu.

Try the following.

```
import next_power
val = next_power.next_power_of_five(126)
print(val)
```

```
import next_power as npow
val = npow.next_power_of_five(44)
print(val)
```

```
from next_power import next_power_of_five
val = npow.next_power_of_five(309)
print(val)
```

It's difficult to overstate how helpful this is in creating cleaner and more readable Jupyter notebooks.

17.3 Using the `random` module and `matplotlib`

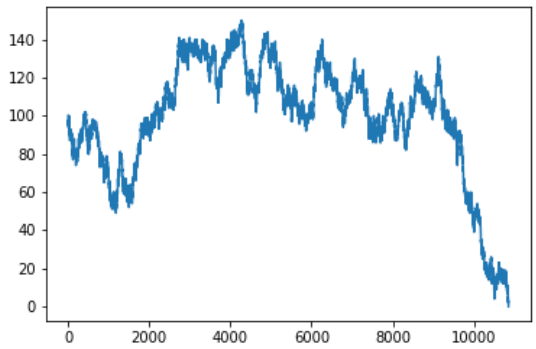
Let's get our hands dirty a bit.

```
import random
random.seed(33) # pick a seed for reproducibility

number = random.randint(0,1)
```

```
# Gambler's Ruin

purse = 100 # starting money

# Keep gambling if you have money
total_gambles = 0
purse_values = [purse]
while purse > 0:
    outcome = random.choice([-1,1]) # win or lose 1 with eve odds
    purse += outcome

    purse_values.append(purse)
    total_gambles += 1
```

We can graph this with `matplotlib`.

```
import matplotlib.pyplot as plt
plt.plot(range(total_gambles+1), purse_values)
plt.show()
```



18 Object Oriented Programming

Reference: Gaddis Chapter 10

So far, our programming has been *procedural*. A program was made of procedures and data might be passed from one procedure to the next. This creates a separation of data and the procedures/operations. As a program grows, this can become more unwieldy. We'll talk about defining classes. While you might get quite far without having to define your own classes, the vocabulary here

is important for anyone who wants to be fluent in Python and understand popular libraries like Pandas.

Object-oriented programming (OOP) centers on creating objects instead of procedures. An object contains both data and procedures. An object's data are its *data attributes*. An object's procedures are its *methods*, which operate on the data attributes. Attributes are like variables and methods are like functions. Bundling data with the code operating on it is called *encapsulation*.

18.1 Classes (G§10.2)

A *class* is code that specifies the data attributes and methods for a particular type of object. A class is like a blueprint and the object is the particular realization. When we previously talked about data types, we were really referencing classes. Run `print(type('Hello, World!'))`. Your output should be `<class 'str'>`.

Classes are defined in the following way. Note, per PEP 8, class names should follow the CapWords convention.

```
import random

class Coin:

    def __init__(self):
        self.sideup = "Heads"
        self.coin = "Quarter"
```

Run `coin = Coin()` and print `coin.sideup` and `coin.coin`.

So a minimal class definition includes the `__init__` *initializer method*. This initializes the objects data attributes. We could instead make these attributes arguments.

```
import random

class Coin:

    def __init__(self, sideup = 'Heads', coin = 'Quarter'):
        self.sideup = sideup
        self.coin = coin
```

Take care to note the last two lines in the block above are necessary to create the `sideup` and `coin` attributes. Now, let's add some methods to our class.

```
import random

# Simulate a coin that can be tossed

class Coin:

    def __init__(self, sideup = 'Heads', coin = 'Quarter'):
        self.sideup = sideup
        self.coin = coin

    def toss(self):
        toss_outcome = random.choice(['Heads','Tails'])
        self.sideup = toss_outcome # Here we change the attribute instead of
    returning a value
```

See what happens now.

```
coin = Coin() # start with a coin that is heads up per class default value
for i in range(100):

    print(coin.sideup)
    coin.toss() # toss the coin
```

Here we add a value-returning method.

```
import random

# Simulate a coin that can be tossed

class Coin:

    def __init__(self, sideup = 'Heads', coin = 'Quarter'):
        self.sideup = sideup
        self.coin = coin

    def toss(self):
        toss_outcome = random.choice(['Heads','Tails']) # local variable just like before in defining functions
        self.sideup = toss_outcome # Here we change the attribute instead of returning a value

    def get_sideup(self):
        return self.sideup
```

Next, we might want to use *hidden attributes*. Before we could externally change `sideup` attribute. Perhaps you don't want that to be possible in this or in another setting. Then you can make that attribute private by including two underscores with the init.

```
import random

# Simulate a coin that can be tossed

class Coin:

    def __init__(self, sideup = 'Heads', coin = 'Quarter'):
        self.__sideup = sideup
        self.coin = coin

    def toss(self):
        toss_outcome = random.choice(['Heads','Tails']) # local variable just like before in defining functions
        self.__sideup = toss_outcome # Here we change the attribute instead of returning a value

    def get_sideup(self):
        return self.__sideup # We need two underscores here too!
```

Now for `coin = Coin()`, you cannot access the `sideup` attribute with either `coin.sideup` or `coin.__sideup`. The `get_sideup()` method becomes necessary to access the private attribute. When making a data attribute private, one might create methods for accessing and changing those attributes. These are called accessor and mutator methods. Or you might call them getters and setters, respectively.

The `__str__` method is designed to indicate an object's emphstate (the attribute values).

```
import random

class Coin:

    def __init__(self, sideup = 'Heads', coin = 'Quarter'):
        self.sideup = sideup
        self.coin = coin

    def __str__(self):
        return "The coin is a " + self.coin + ", and it is " + self.sideup + "."
```

This method is accessed not directly, but by printing the object.

19 Inheritance

Reference: Gaddis Chapter 11

`Inheritance` allows a new class to extend an existing class. This helps with code reusability a bit because we can have super and subclasses.

We might start with a superclass. Here's an example.

```
class Automobile():

    def __init__(self, gas_tank):
        self.gas_tank = gas_tank

    def drive(self):
        self.gas_tank -= 1
```

Now let's make a subclass.

```
class HybridCar(Automobile):

    def __init__(self, gas_tank, battery):
        Automobile.__init__(self, gas_tank)

        # initialize additional battery parameter
        self.battery = 100
```

Try defining `prius = HybridCar(50,100)` and running `prius.drive()`. It should work. Check `prius.gas_tank`. This illustrates basic inheritance of classes.

19.1 Polymorphism (G§11.2)

Now we demonstrate *polymorphism.* A subclass can have the same methods defined as their superclass. The methods override the superclass.

```
class HybridCar(Automobile):

    def __init__(self, gas_tank, battery):
        Automobile.__init__(self, gas_tank)

        # initialize additional battery parameter
        self.battery = battery

```

```
    def drive(self):
        self.gas_tank -= .5
        self.battery -= 1
```

Python includes a handy `isinstance()` function that helps determine if an object is of a certain class or of an instance of a subclass of that class.

```
print(isinstance(prius, Automobile))
print(isinstance(prius, HybridCar))
```

Part VI

Lecture 6

20 IPython and Jupyter

Reference: McKinney Chapter 2

Thus far, we've been using Google Colab, which runs IPython notebooks. Now, we'll use Jupyter. Basically, we're moving off of the cloud.

20.1 IPython Basics (M§2.2)

The Jupyter notebook is an interactive document for code, text, data visualization, and other output. Python's Jupyter kernel uses the IPython system, so all of the IPython basics we'll cover will apply to Jupyter notebooks. Beyond simply executing Python code, IPython comes with enhanced features that make coding easier. I've highlighted a few here. Read through McKinney §2.2 or check the IPython sections from Vanderplas's Python Data Science Handbook.

Tab completion is a feature that allows you to hit tab following some input and then any variables matching the characters will be displayed.

Introspection refers to the ability to place a `?` before or after a variable and executing the line will display helpful information about the object. For a function, you'll be shown the *docstring*.

Notebooks are divided into discrete code cells. You can select multiple with shift and select. Then use `shift-m` to **merge cells**. Use `ctrl-c` to interrupt a command.

There are also certain **magic commands** that begin with a `%`. My favorite are `%%timeit` and `%load`. See McKinney page 29 for a table of magic commands or Vanderplas here.

You can run **shell commands** with `!` or `%` as well. The former creates a subshell, so it cannot be used for changing your working directory. See more here (Vanderplas).

This is on the more complicated end, but here is how I intended to grade your .py files from the midterm.

```
# all submissions are store in folder PyFiles
import os # module for additional string-based shell commands
```

```
from importlib import reload # allows for reloading a module

# ls lists all files in a folder
# Put all file names in a list
files = !ls ~/PyFiles

# Change working directory
%cd ~/PyFiles

for file in files:
    # rename the file
    os.system("mv " + file + " temp.py")
    import temp
    reload(temp) # overwrites previous iteration import

    # grade file
    print(file)
    print(temp.add(2,3)) # all files contain a function named add

    # name the file back
    os.system("mv temp.py " + file)
```

21 NumPy

Reference: McKinney Chapter 4, VanderPlas Chapter 2

NumPy (Numerical Python) is an important library for working with arrays. This shouldn't require any additional installation in Anaconda and is available in Google Colab. NumPy is conventionally imported with the alias `np`: `import numpy as np`.

21.1 The Array (M§4.1)

The NumPy array is important.

```
import numpy as np

empty_array = np.array([])
print(len(empty_array))

zero1 = np.array([0])
print(len(zero1))

zero2 = np.array(0)
#print(len(zero2))

#try_this = np.array(1,2)

two_d = np.array(((1,2),(3,4)))

for array in [empty_array, zero1, zero2, two_d]:
    print(type(array))
```

The array is of a class ndarray.

21.2 Indexing (M§4.1)

Basic indexing is done like for lists. However, arrays can have multiple dimensions.

```
arr = np.arange(10)

print(arr[5]) # the 5th index

# slice
print(arr[5:8])
# reassign
arr[5:8] = 0
print(arr)

print(arr[10]) # error remember we have zero-based indexing
```

Array slices are *views*. The data is not copied and any modifications are transferred to the source array.

```
arr_slice = arr[5:8]
print(arr_slice)

arr_slice = 1 # does nothing bc it writes over the slice with an int
print(arr, arr_slice)

arr_slice = arr[5:8]
print(arr, arr_slice)

arr_slice[:] = 1, 2, 3 # mutates both
print(arr, arr_slice)

arr_slice[:] = -40 # mutates both
print(arr, arr_slice)
```

With multidimensional arrays, there's a bit more to indexing.

```
arr2d = np.array( ( (1,2), (8,9) ) )
```

Thinking of the two-D array as a matrix, the first index place will select the row and the second will select the column.

```
print(arr2d[0])

print(arr2d[0][0])
print(arr2d[0,1]) # these are the same


print(arr2d[:,-1]) # get the last item from each row
```

Check Figure 4-2 in McKinney for a good illustration of slicing two-D arrays.

21.2.1 Boolean Indexing

Recall `arr` from above. We left it with value `arr = np.array([ 0, 1, 2, 3, 4, -40, -40, -40, 8, 9])`. We can index it based on a boolean condition using a syntax `arr[<condition>]`

```
print(arr)

print(arr[arr > 0]) # reduces the array
```

We can combine conditions with logical negations, ands and ors, but *not* with the `not`, `and`, `or` keywords. Use `~` to negate an array of booleans, `&` for and, and `|` for or.

21.3 Functions (McKinney §4.2, 4.3)

Universal functions perform element-wise operations.

```
arr = np.arange(10)

print(np.sqrt(arr))
```

There are also import statistical functions like `np.mean()` and `np.std()` for averaging and finding the standard deviation.

21.4 Linear Algebra (McKinney §4.5)

We can access linear algebra functions using the `numpy.linalg` *submodule*.

As demonstrated above, operators like `*` apply element wise. If we have an array `x = np.array([1,2])`, then `x * x` gives an output of `np.array([1,4])`. Thus, `*` is not the dot product $\cdot$. To get the scalar-valued $x \cdot x = \sum x_i^2$, we need `np.dot(x,x)`. The below illustrates the same with matrix multiplication.

```
x = np.array([[1,0],[0,1]]) # identity
y = np.array([[8,1], [2,3]])

mystery1 = x * y
mystery2 = np.matmul(x,y)
```

You'll find `mystery2 == y` which is what should be expected for matrix multiplication. `mystery1` gives `array([[8, 0],[0, 3]])`, meaning element-wise multiplication was done. Python didn't know you wanted matrix multiplication. You can however call create a specific matrix object with `np.matrix()`. Let's repeat the above.

```
x = np.matrix([[1,0],[0,1]]) # identity
y = np.matrix([[8,1], [2,3]])

mystery1 = x * y
mystery2 = np.matmul(x,y)

print(mystery1 == mystery2)
```

You can invert a matrix with `np.linalg.inv()`, and it will accept either an array or `np.matrix` object.

21.4.1 Regression Exercise

Let's illustrate with a simple linear regression with no intercept. Suppose we have a single independent variable and linear model

$$y = \beta x + \epsilon.$$

Let's simulate some data. We'll assume $\beta = 1$, $\epsilon \sim N(0,1)$, and find $\hat{\beta}$ from simulated data. Recall $\hat{\beta} = (X^T X)^{-1} X^T Y$ so this is a matter of matrix multiplication.

```
import numpy as np

n_obs = 100 # observations
x = np.random.random(n_obs)
epsilon = np.random.normal(0,1,n_obs)
true_beta = 1

y = true_beta * x + epsilon

# make `tall` matrices with rows for each observation
x_matrix = np.matrix(x).T
y_matrix = np.matrix(y).T

# Check on linear algebra and operators
(x_matrix.T * x_matrix)**-1 == np.linalg.inv( np.matmul(x_matrix.T, x_matrix) )
# above actually compares element to element


beta_hat = (x_matrix.T * x_matrix)**-1 * (x_matrix.T * y_matrix)
```

Part VII

Lecture 7

22 Pandas

Reference: McKinney Chapter 5

Pandas was created by Wes McKinney. He's also the author of the text we're using. If you'll work with data in Python, get used to having this at the top of your code.

```
import pandas as pd
```

22.1 Data Structures: Series and DataFrame (M§5.1)

A Pandas *series* is like an array. Unlike lists or numpy arrays, a series has an *index*. If you create a series from scratch, the default index will be numbered from zero. You can also explicitly pass an index. These are constructed much like a numpy array, but with `pd.Series()`. You can also pass a dictionary to create a series, where the keys are the index values.

```
ser1 = pd.Series([10,10,2,20])
ser2 = pd.Series(['squirrel', 100], index = ['animal', 'number'])

dictionary_helper = {'Today': 60, 'Tomorrow': 50, 'Monday': 55}
ser3 = pd.Series(dictionary_helper)


print(ser1)
print(ser2)
print(ser3)

# Compare to Numpy
```

```
import numpy as np
print(np.array(ser1)) # no more index
print(np.array(ser2))

# Compare ser3 to its dictionary
print("Today" in dictionary_helper)
print("Today" in ser3)

print(60 in dictionary_helper)
print(60 in ser3)
```

Correction from Last Lecture: Last time in class, I said that numpy arrays cannot be of mixed type. Examining `np.array(ser2)` demonstrates this is not true. Mixed type can be constructed by specifying the data type as object, `np.array(['s',1], dtype = object)`. Still I don't know a good reason to use NumPy with mixed types. And as the name suggests, NumPy is designed for numeric data.

We can access the series index and values similarly as we'd access a dictionary's keys and values.

```
print(ser3.index)
print(ser3.values)
```

Mathematical operations are automatically aligned based on the series index.

```
wealth = {'Alice': 65, 'Bob': 50}
wealth_series = pd.Series(wealth)

bonus = {"Bob": 10, "Alice": 10}
bonus_series = pd.Series(bonus)

print(wealth_series + bonus_series)

# convert to numpy and add
print( np.array(wealth_series) + np.array(bonus_series) )
```

Null values can arise in a series. An index might be explicitly specified and without any corresponding value, or an operation might not be possible for a particular index.

```
wealth = {'Alice': 65, 'Bob': 50}
wealth_series = pd.Series(wealth, index = ['Larry', 'Alice', 'Debbie', 'Bob'])

bonus = {"Bob": 10, "Alice": 10}
bonus_series = pd.Series(bonus)

new_wealth = wealth_series + bonus_series
print(new_wealth)
```

There are a few methods to help with nulls. The methods `isnull()` and `notnull()` return booleans as the names would suggest.

```
print(new_wealth.isnull())
print(new_wealth.notnull())

print(new_wealth.isnull() | new_wealth.notnull()) # guess what this will be
```

22.2 Series Functionality I

Like with NumPy arrays, you can add two series, multiply two series, etc. You can also add, multiply, etc by a constant.

```
a = bonus_series + 1
b = bonus_series + bonus_series
c = bonus_series * 2 * bonus_series
```

You can also apply NumPy functions that will operate element-wise.

```
a = np.sqrt(bonus_series)
b = np.exp(bonus_series)
c = np.log(b)
```

22.2.1 Apply

Some functions don't automatically apply to sequences element-wise, but you might want them to. The `apply()` method is made for these cases. Pass a function to `apply` and it will be applied at every index in the series.

```
def odd_or_even(x):
    if x % 2 == 0:
        return "Even"
    return "Odd"

ser = pd.Series(range(1,9))

# This gives an error
odd_or_even(ser)

# Use apply
ser.apply(odd_or_even)
```

Anonymous functions are especially useful with the `apply` method.

22.2.2 Anonymous Functions

Anonymous, or lambda, functions are defined without the `def` keyword. They are nameless and can come in handy when needed for a short period. They are often used inside other functions. The follow a syntax like `lambda [argument]: [expression to return]`.

```
example = lambda x: x+1
print(example(-1))
```

```
ser.apply(lambda x: '2' in str(x))
```

22.3 DataFrames (M§5.1

You can imagine a series with multiple columns. That would be a dataframe, `pd.DataFrame`. Below are a few constructions.

```
# construct some DataFrames()
a = pd.DataFrame() # empty
b = pd.DataFrame(ser)
c = pd.DataFrame(ser, ser) # less common
d = pd.DataFrame([ser,ser]) # less common
```

DataFrames are also commonly constructed with a dictionary.

```
data = {'State' : ['KY', 'NY'],
        'Capital' : ['Frankfort', 'Albany']}
df = pd.DataFrame(data)
```

You can also read a CSV with `pd.read_csv()`.

```
atus_df = pd.read_csv("ATUS_activity_2019.csv")
```

The `head()` and `tail()` methods to display the first or last rows. By default, five rows will be selected.

A specific column can be accessed with dict-like notation or by attribute.

```
df['State']
df.State
```

Similarly, a new column can be created with the same dict-like notation.

```
df['Extra Column'] = None
```

Rows and columns can be dropped with the `drop()` method. Columns can also be deleted with `del`.

```
df['Extra Column'] = None
print(df.columns)

del df['Extra Column']
print(df.columns)

df['Extra Column'] = None
df.drop('Extra Column', axis = 'columns', inplace = True)

# axis = 1 also references columns
df['Extra Column'] = None
df.drop('Extra Column', axis = 1, inplace = True)
```

22.3.1 Indexing

You can specify an existing column as the index with `set_index()`. You can also explicitly change the index by accessing the index attribute. You can reset the index with `reset_index()`.

```
print(df.index)

df.index = [1,'clown']

df.set_index("State") # returns a new dataframe
df.set_index("State", inplace = True) # alters df

# go back to numbered index
df.reset_index(inplace = True)
```

A DataFrame can be index with either `loc` or `iloc`. Use `loc` to index by the exact index and column names. Use `iloc` to index by the index and column numbers. An index can contain duplicates, which can complicate the below.

```
# return to State index
df.set_index('State', inplace = True)

a = df.loc['KY', 'Capital]
b = df.iloc[0, 0]

print(a,b)
```

As you could select an entire column with `df['Capital']`, you can select an entire row with `df.loc['KY']` or `atus_df.loc[0]`. Or you can select a subset by passing a list or slicing

```
sub_df1 = atus_df.loc[[0,10,29]]
sub_df2 = atus_df.loc[0:2]
```

22.3.2 Summarizing and Computing Descriptive States (M§5.3)

Let's return to `atus_df`. Let's examine sleep averages and find the person who slept for the longest.

First we can mask to just select the rows where the activity is sleeping.

```
is_sleeping = atus_df.activity_name == 'Sleeping'
sleep_df = atus_df[is_sleeping]

avg_sleep = sleep_df.TUACTDUR.mean()

# even more info
summary_stats = sleep_df.TUACTDUR.describe()

# idxmax gives index with max value
max_row = sleep_df.TUACTDUR.idxmax()

# compare
sleep_df.loc[max_row]
atus_df.loc[max_row]
```

```
a = atus_data.activity_name.value_counts()
b = atus_data.activity_name.value_counts(normalize = True)
```

Part VIII

Lecture 8

23 Application: Primitive Pandas

This section doesn't exactly match anything in the book. We'll apply some of what we learned from McKinney Chapter 5 and consider some old-fashioned ways to loop through a DataFrame.

Consider the `rock_paper_scissors.csv` dataset. Load it using `pd.read_csv`. Each row represents a unique person and their strategy in a game of Rock, Paper, Scissors, where a strategy is just the chance they select either rock, paper, or scissors.

Let's verify that the probabilities sum to one.

1. Do this with a for loop.
2. Do this with the `sum` method.

Let's look for individuals who have very uneven strategies, in the sense that they lean toward any of the three actions with a chance greater than 0.5.

1. Do this with a for loop.
2. Do this with the `max` method.

Create three new columns that are agnostic of the specific rock, paper, or scissors actions and instead give the highest share, the second highest share, and the lowest share.

24 The Basic Join

Reference: McKinney Chapter 8

We will cover merges and joins more in depth in the future, but for now let's consider the special case of joining two DataFrames.

DataFrames have a *join* instance for merging by the index. This requires similar indices and non-overlapping columns.

```
ser1 = pd.Series({'Alice':0.3, 'Bob':0.6})
ser2 = pd.Series({'Alice':0.7})

df1 = pd.DataFrame(ser1)
df2 = pd.DataFrame(ser2)

# This will fail
#joined = df1.join(df2)

# Rename columns
df1.columns = ['Rock']
df2.columns = ['Other']

joined = df1.join(df2)
joined2 = df1.join(df2, how = 'outer')

# examine
joined1 = df1.join(df2)
joined2 = df2.join(df1)
joined3 = df2.join(df1, how = 'outer')

print(joined1)
print(joined2)
print(joined3)

print(joined3.dropna())
```

25 Data Aggregation and Group Operations

Reference: McKinney Chapter 10

25.1 GroupBy

The `groupby` method is fundamental to many data summary tasks. Load `purchase_transactions.csv`.[2] This dataset contains a row for each transaction, with `id` identifying the customer and `item` and `spent` giving the purchased item and spent giving the amount spent (corresponding to a quantity).

It'd be natural for us summarize this data by the individual customer. This requires creating a *GroupBy* object using the `groupby` method.

25.1.1 DataFrame Group By Object

```
grouped = df.groupby('id')
grouped.mean() # try this
```

25.1.2 Series Group By Object

```
grouped = df.groupby('id')['item']
grouped.mean() # try this
```

25.1.3 Group By With Multiple Columns

You can group across multiple dimensions by passing a list into the groupby method.

```
grouped2 = df.groupby(['id', 'item']) # DataFrame groupby object

grouped_df = grouped2.mean() # Creates a DataFrame

grouped_df # inspect
```

This kind of groupby creates a *MultiIndex*, even if you group a series instead of the whole DataFrame. Print `grouped_df.index` to see the following.

```
MultiIndex([(   0,  'apple'),
            (   1, 'butter'),
            (   2,  'apple'),
            (   2, 'butter'),
            (   2, 'orange'),
            (   2, 'turnip'),
            (   3, 'turnip'),
            (   4, 'orange'),
            (   4, 'turnip'),
            (   5,  'apple'),
            ...
            (4994, 'orange'),
            (4994, 'turnip'),
            (4995, 'orange'),
            (4996,  'apple'),
            (4996, 'butter'),
            (4996, 'orange'),
            (4996, 'turnip'),
            (4997, 'turnip'),
            (4998,  'apple'),
```

[2] This is a randomly generated dataset that came from the lifetimes package and then I added extra columns. While more familiar topics might be easier, I would recommend this as a presentation subject for anyone interested in lifetime value calculations.

```
            (4999, 'turnip')],
           names=['id', 'item'], length=8986)
```

While the index of `df` was a list of integers, this is a list of tuples. A row of `grouped_df` is accessed with the standard loc, `grouped_df.loc[(0,'apple')]`.

MultiIndices can complicate your code. You can get rid of the MultiIndex with `unstack`.

```
grouped_df.unstack()
```

Now, the values in the second dimension of the index become columns.

25.2 Pivot Tables

```
df.pivot()
```

A generalization that can handle duplicate values for one index/column pair

```
pd.pivot_table(df)
```

25.3 Crosstabs

```
pd.crosstab(df1.id, df1.item)
```

Part IX

Lecture 9

26 Concat and Append

Reference: McKinney Chapter 8
See Also: VanderPlas Chapter 3

The simplest way to combine two datasets is by concatenating them. Appending one dataset to another can be like a SQL union.

First, there is the `append` method. Consider the two American Time Use Survey datasets, `ATUS_activity_2018.csv` and `ATUS_activity_2019.csv`. It might be natural to combine these datasets if we don't see an important difference between 2018 and 2019. And even if there is an important difference, that could be noted by an extra column indicating the year.

With the `append` method[3], we can simply call `df2018.append(df2019)`. Note this returns a new DataFrame. You might assign this to a new variable if you'd like to work with the combined data.

```
df1819 = df2018.append(df2019)

# Compare number of rows
print(len(df2018), len(df2019))
```

[3] See VanderPlas Chapter 3. Unlike the list append, this does not modify the original object.

```
print(len(df1819))

# Compare number of columns
print(len(df2018.columns), len(df2019.columns))
print(len(df1819.columns))
```

There is also the pandas function `concat`. We can use this to concatenate Series or DataFrames. The objects to be concatenated must be passed as a sequence and there is an optional axis argument.

```
# pd.concat(df2018, df2019) # Doesn't work

df_a = pd.concat([df2018, df2019]) # vertical
df_b = pd.concat([df2018, df2019], axis = 0) # vertical
df_c = pd.concat([df2018, df2019], axis = 1) # horizontal
```

Print out these DataFrames and compare the shapes. Then, inspect the indices. Note that for `df1819`, `df_a`, and `df_b`, the index now contains duplicates. You might amend this with `.reset_index()`. Pandas concatenation preserves indices. You can handle this by

1. Catching duplicates as an error
2. Ignoring the Index
3. Adding MultiIndex keys.

To throw an error if there are duplicates, use the argument `verify_integrity = True`. To ignore the index, specify `ignore_index = True`. Or, to create a MultiIndex, pass an argument `keys = [2018, 2019]` where `keys` could more generally be any list that gives a unique key for each input to the concatenation.

Finally, there is also the `join` argument for `concat()` which can be used when the DataFrames don't share all of their columns. Use `join = 'inner'` to return just the common columns, and use `join = 'outer'` (also the default) to return all columns.

26.1 Application: What precedes sleeplessness?

Concatenation can be useful when you want to add columns that give values from the previous row. As an analyst, you might want to compare a row event with what took place before. We can do this with the help of the `shift` method.

```
df1819.reset_index(drop = True, inplace = True) # Clean index

shift_df1819 = df1819[['TUCASEID','activity_name']].shift() # pushes every row
    forward by default
shift_df1819.columns = ['prev_TUCASEID', 'prev_activity_name']

df1819 = pd.concat([df1819,shift_df1819], axis = 1)

df1819.head()
```

Then,

```
same = df1819.TUCASEID == df1819.prev_TUCASEID
sleepless = df1819.activity_name == 'Sleeplessness'
sleeping = df1819.activity_name == 'Sleeping'
```

Compare `df1819[same & sleepless].prev_activity_name.value_counts(normalize = True)` and `df1819[same & sleeping].prev_activity_name.value_counts(normalize = True)`.

27 Merge

We previously looked at the pandas join, which merged DataFrames based on their indices. Now, we will consider merges more generally, where we can merge based on column values.

A merge can be accomplished with a `.merge()` method, `df1.merge(df2 ... )` or with the pandas merge function, `pd.merge(df1, df2, ...)`.

First, let's consider `pd.merge` and let's load the 2018 ATUS data files. These DataFrames share just one column, `TUCASEID`.

```
activity = pd.read_csv("ATUS_activity_2018.csv", index_col = 'Unnamed: 0')
resp = pd.read_csv("ATUS_respondent_2018.csv", index_col = 'Unnamed: 0')

merge1 = pd.merge(activity, resp)
```

We didn't specify a column to merge on, so the merge is automatically done on the common column. However, it is better to specify using the `on` argument. This is to follow the principle of coding, "Explicit is better than implicit."

As we could use the `verify_integrity` argument in concatenation, we can use a `validate` argument to throw an error if Python doesn't find our expected behavior in the merge. Here, we have a many-to-one merge, because a single TUCASEID appears multiple times in the left dataset, `activity`, and just once in the right dataset, `resp`.

```
try:
    pd.merge(activity, resp, on = 'TUCASEID', validate = 'many_to_one')
except Exception as e:
    print(e)

try:
    pd.merge(activity, resp, on = 'TUCASEID', validate = 'one_to_many')
except Exception as e:
    print(e)

try:
    pd.merge(activity, resp, on = 'TUCASEID', validate = 'one_to_one')
except Exception as e:
    print(e)
```

Merges can also be done on multiple columns. Here we create (fake) supplemental data to be added.

```
activity_supplement = activity[['TUCASEID','TUSTARTTIM']]
activity_supplement.loc[:,'is_alone'] = np.random.choice([True, False],len(
    activity_supplement))

# Jumble dataframe to a simple index join or concat is possible
activity_supplement = activity_supplement.sample(len(activity_supplement)) #
    samples without replacement to shuffle
activity_supplement.reset_index(drop = True, inplace = True)

# Merge
```

```
pd.merge(activity, activity_supplement, on = ['TUCASEID', 'TUSTARTTIM'], validate
    = 'one_to_one')
```

Datasets can also be merged on differently named columns.

```
activity_supplement.columns = ['a', 'b', 'c']
pd.merge(activity, activity_supplement, left_on = ['TUCASEID', 'TUSTARTTIM'],
    right_on = ['a','b'])
```

Finally, there are left, right, inner, and outer joins. These can be specified with `how`.

Consider the following

```
pd.merge(activity, activity_supplement.head(), left_on = ['TUCASEID', 'TUSTARTTIM'
    ], right_on = ['a','b'])
```

What results is a DataFrame of length. By default, pandas does an inner join.

```
supplement2 = activity_supplement.head()
supplement2.columns = ['TUCASEID', 'TUSTARTTIM', 'is_alone'] # name back

# Create a TUCASEID not in the activity dataset
supplement2.loc[0, 'TUCASEID'] = "Uncle Milton"


df_inner = pd.merge(activity, supplement2, on = ['TUCASEID', 'TUSTARTTIM'],  how =
     'inner')

df_outer = pd.merge(activity, supplement2, on = ['TUCASEID', 'TUSTARTTIM'], how =
    'outer')

df_right = pd.merge(activity, supplement2, on = ['TUCASEID', 'TUSTARTTIM'], how =
    'right')

df_left = pd.merge(activity, supplement2, on = ['TUCASEID', 'TUSTARTTIM'], how = '
    left')
```

Inspect each. These DataFrames are all unique!

28 Matplotlib I

Reference: VanderPlas Chapter 4

Matplotlib is the standard library for visualization in Python. There are two interfaces: a MATLAB-inspired interface and an object-oriented interface. That is, you can create plots with either of this code styles. Here, we will work with the simpler MATLAB style.

The standard imports and shorthands are as follows.

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

Here's a minimal working example for a plot.

```
import numpy as np

x = np.linspace(0, 10, 100)
plt.plot(x, np.sin(x))
```

```
plt.plot(x, np.cos(x))

plt.show()
```

Using `plt.show()` is not always necessary in a Jupyter environment. See VanderPlas Chapter 4 for a slightly more technical discussion. It's sufficient to think of `plt.show()` as "finishing" the plot. This can be helpful when you want to create multiple plots in one code block.

Compare the following two programs.

```
x = np.linspace(0,10,100)

for i in range(0,10):

    y = np.ones(len(x)) * i
    plt.plot(x, y)
    plt.show()
```

```
x = np.linspace(0,10,100)

for i in range(0,10):

    y = np.ones(len(x)) * i
    plt.plot(x, y)

plt.show()
```

The only difference is the indentation of `plt.show()` Observe the first creates ten different plots. The second creates just one plot.

28.1 Saving Figures

You can use `plt.savefig` to save a plot. This must go before `plt.show()`. There are many supported filetypes. I am partial to png and pdf. A `dpi` argument can be used to set the image resolution.

```
x = np.linspace(0,10,100)

for i in range(0,10):

    y = np.ones(len(x)) * i
    plt.plot(x, y)

plt.savefig('example_figure.png', dpi = 300)
plt.show()
```

28.2 DataFrame and Series Plot Methods

There are Pandas methods that create Matplotlib objects. You'll get pretty far with `.plot()` and `.plot.bar()` or `.plot.barh()`.

Let's return to our sleeplessness application. We can call `.plot.bar()` for a quick bar graph.

```
top_activities_before_sleeping = df1819[same & sleeping].prev_activity_name.
    value_counts(normalize = True).head()
top_activities_before_sleepless = df1819[same & sleepless].prev_activity_name.
    value_counts(normalize = True).head()

```

```
4
5 top_activities_before_sleeping.plot.bar()
```

These don't quite like in the first example where we could keep adding to a plot. For example, the below does not create a side-by-side barplot or even create accurate labels on the x-axis to account for the differing indices.

```
1 top_activities_before_sleeping = df1819[same & sleeping].prev_activity_name.
      value_counts(normalize = True).head()
2 top_activities_before_sleepless = df1819[same & sleepless].prev_activity_name.
      value_counts(normalize = True).head()
3
4 # This is bad!
5 top_activities_before_sleeping.plot.bar(color = 'red', alpha = 0.5)
6 top_activities_before_sleepless.plot.bar(color = 'blue', alpha = 0.5)
```

For a side-by-side plot, it is better to use a DataFrame instead of a Series.

```
1 a = pd.DataFrame(top_activities_before_sleeping)
2 a.columns = ['sleeping']
3
4 b = pd.DataFrame(top_activities_before_sleepless)
5 b.columns = ['sleepless']
6
7 a.join(b, how = 'inner').plot.bar()
```

Part X

Lecture 10

29 Data Cleaning and Preparation

Reference: McKinney Chapter 7

29.1 Missing Data

For missing numeric data, Pandas uses the value `NaN`, for Not a Number. If you want to manually insert such a value, use `np.nan`. Such missing data is refferred to as NA. The Python `None` value is also treated as NA.

NA handling methods:

- `dropna`
- `fillna`
  - `ffill`
  - `bfill`
- `isnull`

- `notnull`

29.2 Data Transformation

Data might also be transformed by handling duplicates, remapping values, discretization, binning, removing outliers, sampling, or creating dummy variables (one-hot encoding).

29.2.1 Binning

We can bin values by using `pd.cut`.

```
data = np.random.normal(0,1,size = 1000)
df = pd.DataFrame(data)

bins = [-10 ** 9, -1.96, 0, 1.96, 10 ** 9]

cats = pd.cut(data, bins)
df['bin'] = cats

pd.value_counts(cats)
```

You can also specify labels.

```
bin_labels = ['extremely negative', 'negative', 'positive', 'extremely positive']
cats2 = pd.cut(data, bins, labels = bin_labels)

df['bin_label'] = cats2
```

29.2.2 Dummy Variables

Use `pd.get_dummies` to convert a categorical column into several columns of dummy variables. Using the same `df` from above,

```
dummies = pd.get_dummies(df['bin_label'])

df_with_dummy = df.join(dummies)
```

29.2.3 Map and Apply

Here's a relevant Stack Overflow post.

You can use `apply` on a DataFrame or a Series. You can use `map` on a Series.

```
# This works
dummies.apply(np.sum) #, axis = 1)

# This fails
dummies.map(np.sum)

# This works
dummies['extremely_negative'].apply(lambda x: 2*x)
# Also works
dummies['extremely_negative'].map(lambda x: 2*x)
```

29.3 String Manipulation

There are many vectorized (fast) string methods. They can be used on dataframes by including `.str` at the end of a series before applying the method.

Application: Wikipedia Game Data. Count the number of clicks.

```
df = pd.read_csv('WikipediaGame.csv')

df['contains_backpage'] = df.path.str.contains('<')

df['clicks'] = df.path.str.count(";")


# Get starting and ending
df['starting_page'] = df.path.apply(lambda x: x.split(";")[0])

df['destination_page'] = df.path.apply(lambda x: x.split(";")[-1])
```

30 Matplotlib II

Reference A: VanderPlas Chapter 4
Reference B: McKinney Chapter 9

30.1 Object-Oriented Interface

In addition to the MATLAB style interface, matplotlib can be used with an object-oriented style. Here, a plot is built around *figure* and *axis* objects.

```
x = np.linspace(0,10,51)

fig, ax = plt.subplots()

# Call plot() on the axis
ax.plot(x, np.sin(x))

# plt.show() # show is a function, not a method on ax or fig
```

You might also create the figure and axis objects directly, without `subplots`. You can use `subplots` to create multiple subplots within the figure.

```
fig, ax = plt.figure(), plt.axes()

# Call plot() on the axis
ax.plot(x, np.sin(x))

plt.show()


# Multiple Subplots

# ax is a tuple for two different axes
fig, ax = plt.subplots(2)

# Call plot() on the axis
ax[0].plot(x, np.sin(x))
```

```
ax[1].plot(x, np.sin(x), color = 'tomato')

plt.show()
```

In the last example above, two subplots are created. If you save this, note this creates a single image file, not one for each subplot.

30.1.1 Further Customization

There are many avenues of customization available. Again, I recommend Chapter 4 from VanderPlas as a reference. An exhaustive rundown would be tedious. I will highlight legends and custom axis ticks.

```
fig = plt.figure(figsize=(6,4))
ax = plt.axes()

x = np.linspace(0, 1, 2)
ax.plot(x, x + 0, label='line 0')
ax.plot(x, x + 1, label='line 1')
ax.plot(x, x + 2, label='line 2')
ax.plot(x, x + 3, label='line 3')
ax.legend(fontsize=12, frameon=True, facecolor='w')
```

We can customize the legend even further, to get it outside of the plot by specifying the location and adjusting the `bbox_to_anchor`.

```
fig = plt.figure()
ax = plt.axes()

for i in range(5):
    line, = ax.plot(x, i * x, label='$y = %ix$'%i)

# Put a legend below current axis
ax.legend(loc='lower center', bbox_to_anchor=(0.5, -0.3),
          fancybox=True, shadow=True, ncol=5)

plt.tight_layout() # prevents anything from being cut off when saving
plt.show()
```

Next, you might notice that matplotlib is automatically setting the ticks shown on the axes. You can customize these, both with the location of the ticks and their labels.

```
fig, ax = plt.subplots()
x = np.linspace(0,10,100)

ax.plot(x, np.cos(x))

# Modify the x-axis
ax.set_xticks([0, np.pi, np.pi*2, np.pi*3])
ax.set_xticklabels([0, "$\pi$", "$2\pi$", "$3\pi$"])

# Add a vertical line for the heck of it
ax.axvline(np.pi, 0, 1, linestyle = 'dashed')

plt.show()
```

30.2 Contour Plots

Matplotlib can be tricky to learn. For a basic graph that you'll only create once, it might be faster to use Google Sheets. Of course you can do more with Python. Contour plots are one such functionality that help justify the price of matplotlib.

Suppose you work for a music-streaming company that streams only Drake and Kanye West. You want to visualize estimated lifetime values for customers depending on how much of each artist they listen to.

```
# Generate Data
x = np.linspace(0,10,11)

def f(x,y):
    return np.sqrt(x) + 3 * np.sqrt(y) + np.random.normal(1,.1)

X, Y = np.meshgrid(x,x)
Z = f(X,Y)

# Make Plot
plt.contourf(X, Y, Z, 20, cmap = 'coolwarm')

plt.xlabel("Drake")
plt.ylabel("Kanye")
plt.title("LTVs")

plt.colorbar()
plt.show()
```

Looking at the graph, you'll notice that more of either artist is good. However, Kanye seems to have higher marginal impact, and from the curvature, you can see that variety is beneficial because the contours are bowed inward.

30.3 GridSpec

For irregular plot grids, `GridSpec` is your friend. You can specify a grid with some number of rows and columns and spacing between them. For example, `grid = plt.GridSpec(2, 3, wspace = 1, hspace = 0.3)`. Then, you can specify subplot locations using the typical slicing syntax. For example, `plt.subplot(grid[0,0])`. Or you can create an axis object for a subplot with `ax = fig.add_subplot(grid[0,0])`.

```
import matplotlib.gridspec as gridspec

data = 'nyc_data.csv'
df = pd.read_csv(data)

# Plot it!
fig = plt.figure(figsize=(12,6)) #
spec = gridspec.GridSpec(ncols=4, nrows=2, figure=fig, wspace = 0.5, hspace = 0.2)

f2_ax1 = fig.add_subplot(spec[0, 0:3])
f2_ax1.plot(df['pickup_longitude'], df['pickup_latitude'], linestyle='None',
    marker='.', alpha=0.5)

f2_ax2 = fig.add_subplot(spec[0, 3:4])
f2_ax2.hist(df['pickup_latitude'], orientation='horizontal', bins=40)

```

```
f2_ax3 = fig.add_subplot(spec[1, 0:3])
f2_ax3.hist(df['pickup_longitude'], bins = 40) #, orientation='horizontal', bins
    =40)
f2_ax3.invert_yaxis()
```

Part XI

Lecture 11

31 Time Series and Datetime

Reference: McKinney Chapter 11
Reference: VanderPlas Chapter 3

When working with dates, the built-in `datetime` and `dateutil` modules are useful.

31.1 Datetime

You can handle a datetime timestmap with the `datetime` type. When the time component is a distraction, you can use `date` objects instead.

```
import datetime as dt

some_dtime = dt.datetime(year = 2000, month = 1, day = 1, hour = 0, minute = 0)
right_now = dt.datetime.today()

some_date = dt.datetime(year = 2000, month = 1, day = 1)
converted_date = right_now.date()
today = dt.date.today()
```

A `timedelta` object is useful for dealing with a duration of time, when subtracting datetimes for example.

```
class_start = dt.datetime(2020,12,5,12,10,0)
class_end = class_start + dt.timedelta(hours = 1, minutes = 50)
```

The datetime `strptime` method is very useful in string conversion. It works with a date string as its first argument and a format as a second argument.

For example, `dt.datetime.strptime('2020-01-01', '%Y-%m-%d')` returns a datetime object for January 1st, 2020.

31.2 Dateutil

Dateutil offers, among other things, a parser and the ability to use a relative time delta.

```
from dateutil import parser
date = parser.parse("9th of December, 2019")
print(date)

from dateutil.relativedelta import relativedelta

```

```
other_date = date + relativedelta(months = 2)
print(other_date)

# Consider how this works with leap year
next_year1 = date + relativedelta(years = 1)
next_year2 = date + dt.timedelta(days = 365)
```

31.3 Pandas and Numpy

Pandas offers an efficient `Timestamp` object and NumPy offers an efficient `datetime64`. For NumPy, the tradeoff is they are less flexible than datetime objects. Pandas offers something closer to the best of both worlds.

Notice the accessing a NumPy array from a Series will convert datetime objects into NumPy's datetime64.

```
ser = pd.Series([dt.datetime.today()])
date = ser.values[0]
print(date)

# date + dt.timedelta(days = 1)  # error
```

Here, we parse a date into a Pandas Timestamp and use a datetime timedelta.

```
date = pd.to_datetime("4th of July, 2015")
print(date)
print(type(date))

date + dt.timedelta(days = 1) # Works
```

31.4 Rolling Means

```
dates = pd.date_range(start='2020-12-05', end='2021-12-05', freq='1D')

df = pd.DataFrame(index = dates)

df['stock_price'] = range(len(dates)) + np.random.normal(0,10,len(dates))

plt.plot(df['date'], df['stock_price'])
plt.show()


## Rolling Function

data = df.stock_price.rolling(30, center = True)

data.mean().plot()
plt.show()
```

Try it with other window sizes.

```
data = df.stock_price.rolling(365, center = True)
data.mean().plot()
plt.show()

data = df.stock_price.rolling(100, center = True)
data.mean().plot()
plt.show()
```

31.5 Application 1

Load the NYC taxi data and look at the distribution of ride lengths.

```
df = pd.read_csv('nyc_data.csv')

df.head(1)
```

```
df['pickup_datetime'] = pd.to_datetime(df['pickup_datetime'])
df['dropoff_datetime'] = pd.to_datetime(df['dropoff_datetime'])

df['duration'] = df['dropoff_datetime'] - df['pickup_datetime']

# Make histogram with logarithmic y axis
df['duration'].dt.total_seconds().hist(log = True)
```

31.6 Application 2

Load the `ATUS_activity_2019.csv` dataset. Create a new datetime time object column from the `TUSTARTTIM` column.

31.6.1 Method 1

```
import datetime
import pandas as pd

df = pd.read_csv("ATUS_activity_2019.csv")

# Parse the time
df['time_col'] = df['TUSTARTTIM'].map(lambda x: dt.datetime.strptime(x, "%H:%M:%S"
    ))

# This sets everything to a datetime object from January 1900
# Convert to time with the time method

df['time_col'] = df.time_col.map(lambda x: x.time())
```

32 Efficient Code

An important word in writing efficient code is *vectorization*. When Python can act on a vector instead of single value after single value, fewer operations are required and speed improves. This will not be a comprehensive look at how to write efficient code, but here are a few areas where you can speed your code.

32.1 Iteration Over DataFrames

If you'd like to iterate over the *rows* of a DataFrame, you should know of the `iterrows` and `itertuples` methods.

32.2 `loc`, `iloc`, `at`

`iloc` is generally faster than `loc`. However, if you want only a single value, `at` or `iat` is generally the fastest.