COLLEGES OF NANOSCALE SCIENCE + ENGINEERING • SUNY POLYTECHNIC INSTITUTE •

NNSE 784
Advanced Analytics Methods

Instructor: F Doyle (CESTM L210)

MW 4:30 – 5:50, NFN 203

Slide Set #1
Course Introduction
Python and Jupyter Notebook Overview

What this course **is not:**

- This is not a statistical theory course. We are going to focus on some of the general concepts and applications in statistics. We are not going to do “deep dives” into the underlying math.
- Similarly, this is not a machine learning course. We will cover some machine learning models, their core concepts and how to apply them to analyze our data, but we will not get deep into the weeds of their implementations.
- This is not a software engineering course. We will learn some basic Python programming and use it to invoke tools to perform high level data analyses.

What this course **is:**

- This course is intended to provide general understanding of some fundamental statistical concepts and methods
- This course will demonstrate how to apply these using popular Python based analytical platforms
- This course will introduce more advanced, machine learning models for analysis and demonstrate their application
- This course will introduce time series data and analysis and use the Seeq analytical platform

General Course Overview

- Introduction to Python Language and Jupyter Notebook Environment
- Statistics (primer/refresher)
  - Descriptive/Inferential Statistics
  - Distributions
  - Hypothesis Tests
- Time Series Data Analysis
- Machine Learning Models
  - Regression vs Classification
  - Linear Regression and Logistic Regression
  - Clustering
  - Decision Trees
  - Support Vector Machines
  - Multi-layer Perceptron

NNSE 784 Advanced Data Analytics Methods Fall 2023

Instructors: Francis Doyle
Email: doylef@sunypoly.edu Office: L210 CESTM

Time: TBD

Location: TBD

Description: An introductory course for data analysis using Python and SeeQ. This course aims to introduce exploratory data analysis, basic concepts of applied statistics, regression analysis, data visualization and a primer in time series data analysis using SeeQ Workbench. A hands-on approach will guide most lessons using common Python libraries (NumPy, SciPy, Matplotlib, pandas, etc.) in a Jupyter notebook environment. The course takes a task-oriented path presenting material in a logical progression from data acquisition, through results presentation. Where possible, techniques will be taught using real data from SUNY Poly labs and processes (or collected from literature sources). Topics include Data Wrangling/Cleaning, Exploratory Data Analysis, Basic Applied Statistics, Regression Analysis and Time Series Data Analysis with SeeQ Workbench. Students will also be given a high-level conceptual introduction to machine learning approaches. Final projects will provide students with the option of choosing their own data for detailed analysis using the techniques covered in the course. Course goal is to provide graduates with a foundation of in-demand analysis skills using common industry tools. Prior Python experience is not required.

Learning Outcomes: Students will….

- ➢ *Gain a conceptual understanding of fundamental tools and techniques in data analysis, and apply their knowledge of the tools by using them to evaluate relevant data and present results in a report format.*
- ➢ *Demonstrate the ability to apply appropriate analysis that evaluates real experimental data.*
- ➢ *Demonstrate hands-on experience by applying the covered concepts in Python and SeeQ.*

Textbook and software:

Python for Data Analysis: Data Wrangling with pandas, NumPy, and Jupyter 3rd Edition
Wes McKinney
ISBN **109810403X**
Course handouts and reading material from the scientific literature will be assigned to supplement the suggested text.

Python/Jupyter Notebooks – *freely available open source software*
SeeQ – *access provided free to students by SeeQ Corporation*

Office hours: TBD and as requested

Grading: Course grade determined by programming assignments (50%), five quizzes (25%) and a final project (25%).

Syllabus: See next page.

Accommodations for Students with Disabilities registered at SUNY Polytechnic Institute

In compliance with the Americans with Disabilities Act of 1990 and Section 504 of the Rehabilitation Act, SUNY Polytechnic Institute is committed to ensuring educational access and accommodations for all registered students seeking access to meet course requirements and fully participate in programs and activities. Students with documented disabilities or medical conditions are encouraged to request these services by registering with the Office of Disability Services. For information related to these services or to schedule a virtual appointment, please contact the Office of Disability Services using the information provided below and visit https://sunypoly.edu/student-life/diversity-inclusion/disabilities-services.html

Megan Wyett Lennon, Interim Director
Office of Disability Services
ds@sunypoly.edu
(315) 792-7170
Peter J. Cayan Library, L145

Albany Campus
Suite 309, Students Services Office, NanoFab South

Proposed Course Outline

- Introduction to Course, Python and Jupyter Notebook environment
- Data acquisition, access and basic descriptive statistics using pandas data frames
- Descriptive statistics and population parameters
- Data cleaning and preparation
- Data wrangling
- Sampling, estimates and confidence intervals
- Hypothesis testing, p-values and the Student's t-test
- Analysis of variance (ANOVA) one and two-way
- Post Hoc test for ANOVA using Tukey's HSD
- Multiple Testing Correction
- Simple and multiple linear regression, predictions
- Polynomial regression, overfitting and underfitting
- Data visualization with matplotlib and Seaborn
- Principal Component Analysis
- Introduction to time series data
- Introduction to Seeq Workbench. Importing data, organizing and navigating view. Seeq knowledgebase
- Seeq conditions for identifying events and periods of interest. Quantification in Seeq
- Data visualization and documentation in Seeq
- Overview of machine learning techniques – part 1
- Overview of machine learning techniques – part 2
- Overview of machine learning techniques – part 3

Lecture Overview

- Tools we will be using
  - Python libraries
  - Analysis environments
- Some background on general programming and Python
- Seeq Data Lab / Jupyter Notebook
- Basic Python usage examples
  - Strings and output
  - Other simple “built in” types
  - Advanced built in data types
  - Control structures
  - Functions

Development Environments

Jupyter notebooks – This is a web based (can be run locally, but is accessed via a browser), interactive development and execution environment that can also be used as a presentation format. It is recommended that you go to www.anaconda.com and install the free download of Anaconda which includes Jupyter. This is the primary environment we will use in the class, and if you are unable to access the Seeq (below) server for any reason, you will be able to work on most of the material via your personal machine. Be aware that if you are using Anaconda for commercial endeavours, there are licensing requirements that differ from academic use. Jupyter may be installed without Anaconda, but it is more complicated.

Seeq is also a web based software designed for analysis of time series data. Seeq is proprietary but the company has donated access for CNSE via a cloud based instance at https://sunypoly.seeq.tech . There are three primary tools bundled in Seeq; Workbench, Organizer, and Data Lab. Data Lab is a customized Jupyter notebook environment. Workbench is the general interface tool for time series data analysis. All of the exercises in this class should be able to be performed in Seeq.

Libraries

NumPy – n-dimensional arrays and math functions (linear algebra routines, Fourier transforms, etc.). Considered a fundamental package for scientific computing. Other libraries we will use are built on top of it.

SciPy – provides algorithms for optimization, integration, interpolation, eigenvalue problems, algebraic equations, differential equations, **statistics**, etc.

pandas – data structures and operations for manipulating numerical tables and time series. The pandas DataFrame is a core data structure that will be used for much of the analyses we perform in this class.

scikit-learn (aka sklearn)– machine learning library for classification, regression, clustering, dimensionality reduction, etc. This library includes support for decision trees, support vector machines, neural networks and other ML models.

Libraries - continued

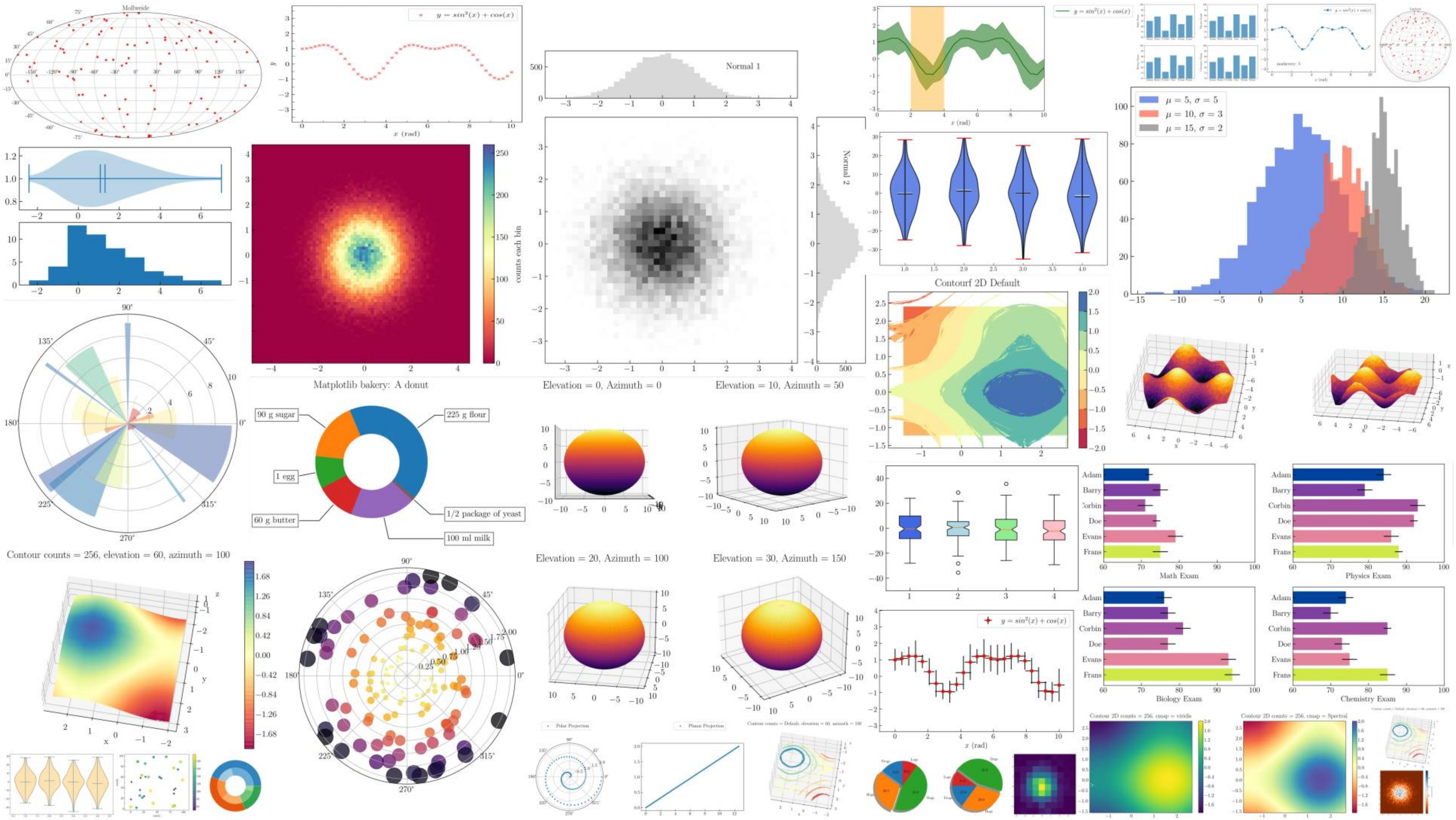
Data Visualization – “A (good) plot is worth a million data points”

Matplotlib – provides support for creating static, animated, and interactive visualizations of data.

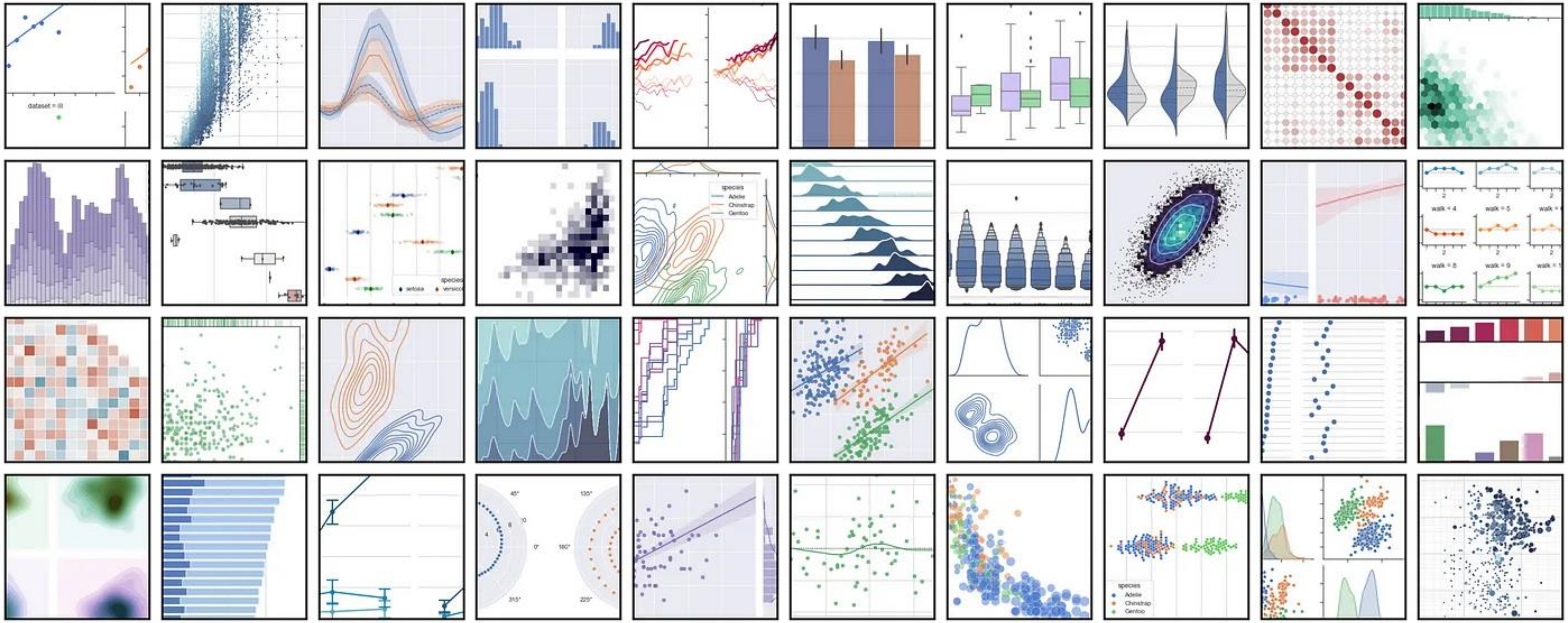
seaborn – based on matplotlib. Provides a high level interface for drawing high quality statistical graphs.

Both of these libraries allow you to produce excellent quality visualizations of your data with minimal effort in coding.

Matplotlib – Example Graphs



Seaborn – example graphs



Python/programming

- At it’s simplest, a program is simply a sequence of instructions specified by a programmer that a computer will execute.
- The term “program” covers an enormous range of possibilities. Think of the word “building” as applied to both a garden shed and a semiconductor fab with all of it’s mechanical systems.
- Our goal isn’t to build the equivalent of a fab. We need to understand some basics to invoke the capabilities provided by the libraries that we will be using (pandas, etc.).

Python/programming - continued

- Interpreted (versus compiled) scripting language.
- Full programs (scripts) can be executed directly, or the interpreter may be run in an interactive mode where each line is executed as it is entered
- Widely considered one of the easier languages for a novice to “pick up” quickly
- Jupyter notebooks is an interactive Python environment (though complicated scripts can be written and executed in this setting)
- **Do not be intimidated if anything seems confusing**. You will be given example code for every problem we cover and I will be available to help you with anything you get stuck on. This class is not going to focus on minutiae. My goal is that you come out of it with a solid familiarity of the concepts and resources to apply them in the future, **not** that you memorize some underlying formula that you’ll forget a month after the course ends.

Python/programming - *continued*

- We will do a quick overview of some important aspects of the Python language in this slide set, but will predominantly "learn by doing" as we apply relevant tools through the length of the course.
- Regardless of programming language, there are some core aspects that are generally found in all
  - Primitives (simple "built ins" in Python) – the simplest data types the language/hardware supports, such as:
    - Integers (-1,0,1,2,etc.)
    - Floating points (5.01326)
    - Strings (e.g., "Some text in a program")
    - Boolean (i.e., True or False)
  - Data Structures, such as
    - Lists
    - Tuples
    - Dictionaries
    - Sets
    - User, or package/library defined

    We will describe these later
  - Variables – these are named containers whose contents can change during program flow. In Python, the type of the variable can also change dynamically
  - Control structures/loops – determine execution flow
  - Functions/methods/subroutines – discrete portion of code to accomplish a specific task and may or may not return a value

A note on data structures:

The importance of data structures to computing can not be overstated.

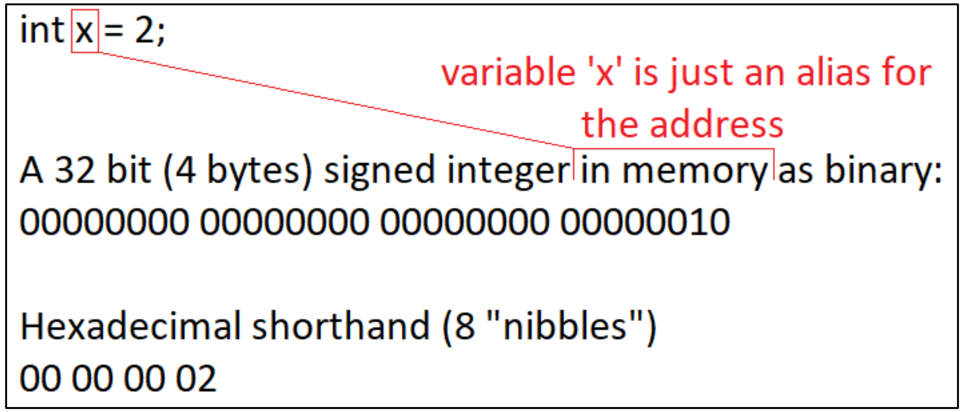
One way to think about them is that they organize the appropriate data in a particular way to accomplish specific tasks. Often, they are accompanied with special logic (functions/methods) that help to accomplish these tasks.

Think about your daily life and how your home is organized into rooms and storage containers.

Think about how much harder your routine would be if every single belonging was just piled into a single heap, or randomly distributed around existing rooms.

TANGENT – you are not expected to remember this for any quiz, etc.

Python “built in” simple data types VS Primitives



int x = 2;

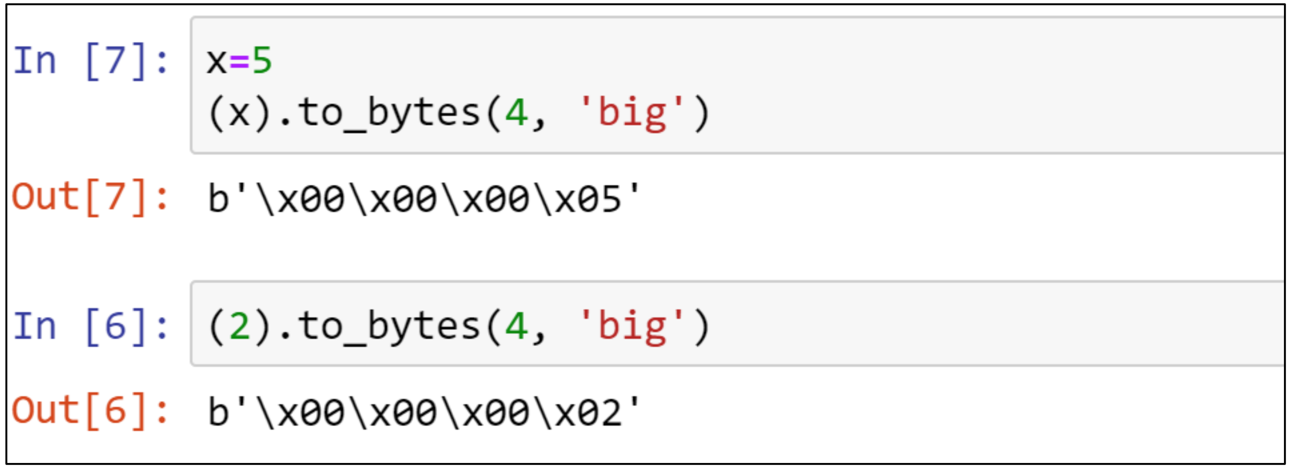
variable 'x' is just an alias for the address

A 32 bit (4 bytes) signed integer in memory as binary:
00000000 00000000 00000000 00000010

Hexadecimal shorthand (8 "nibbles")
00 00 00 02

Traditional, compiled languages (e.g., C, C++, java) treat things like integers as true primitives. The variable is understood to simply point to the address of the first byte in memory of an encoded value that based on the type (e.g., ‘int’ vs ‘long’) will occupy a certain number of subsequent number of bytes.

In Python, EVERYTHING is an object. An ‘int’ variable isn’t pointing to just a binary representation of that integer but rather to an object (an object is a particular type of complex data structure). In fact, even the int literal (without variable assignment) is understood by the interpreter to be an int object.



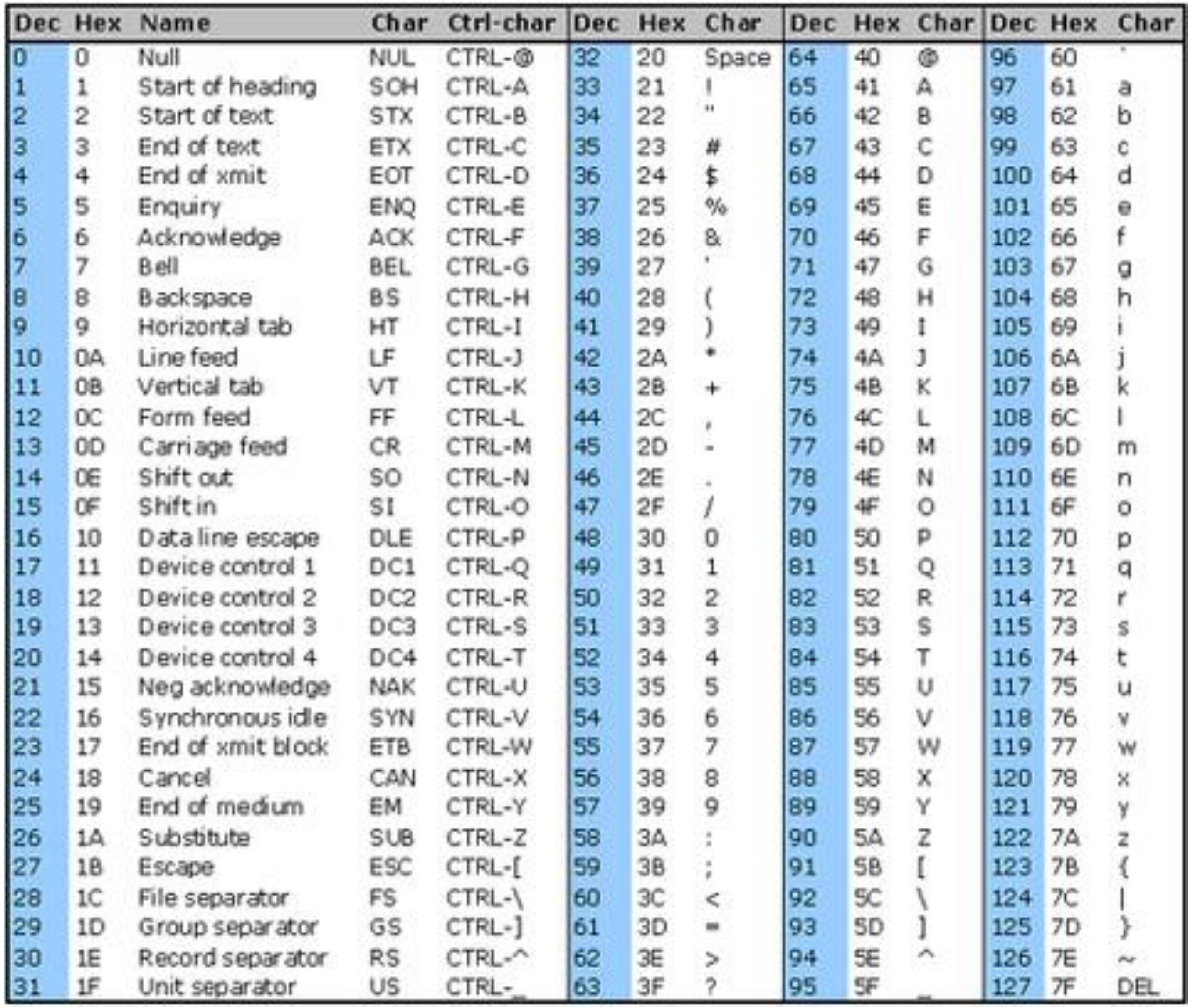
```
In [7]:  x=5
         (x).to_bytes(4, 'big')

Out[7]:  b'\x00\x00\x00\x05'


In [6]:  (2).to_bytes(4, 'big')

Out[6]:  b'\x00\x00\x00\x02'
```

TANGENT – you are not expected to remember this for any quiz, etc.

A Note on Encoding

Computer memory is binary. All data, be it an integer, a floating point, a string, etc. have to be represented (i.e., "encoded") in this format.

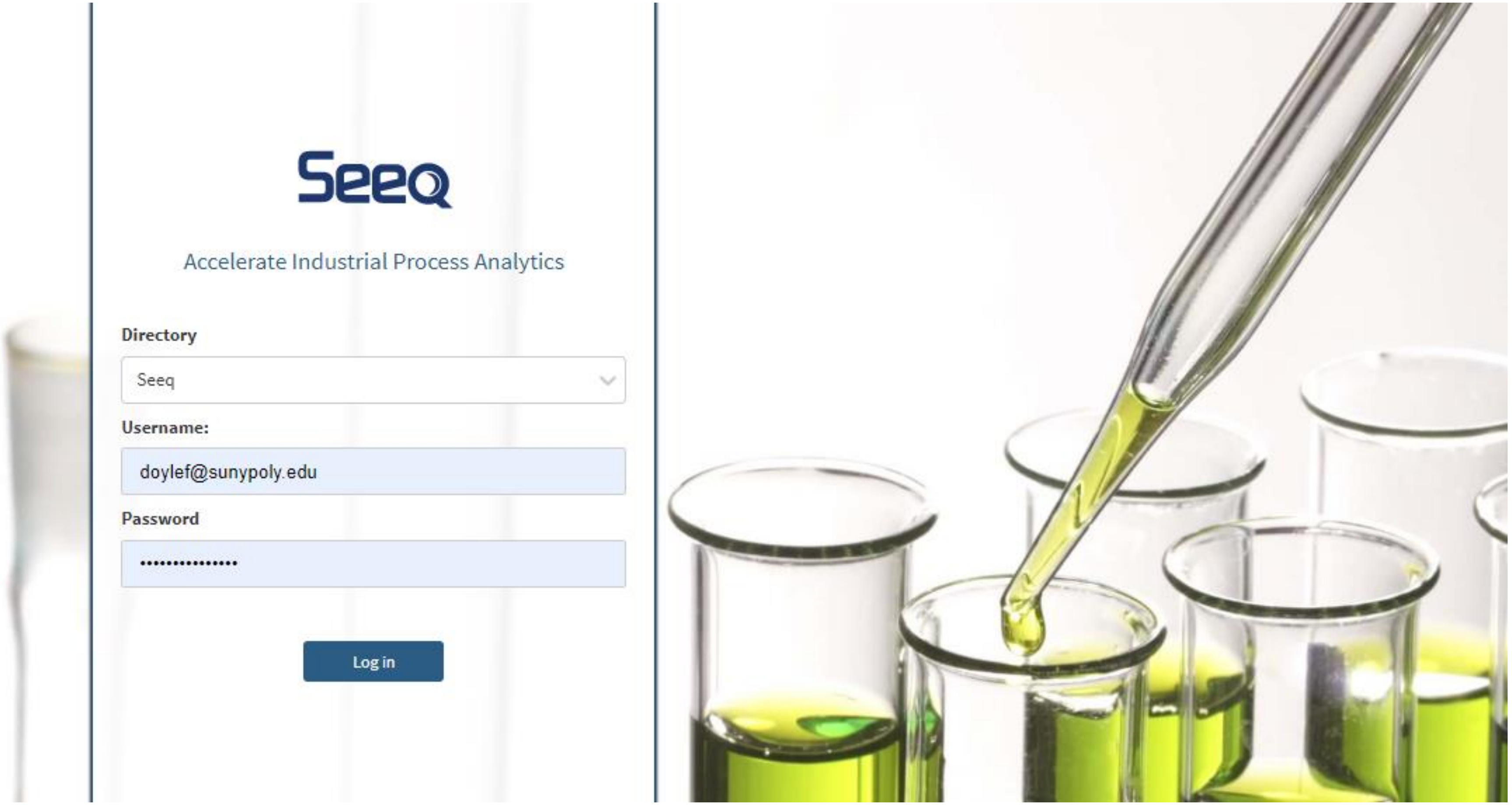
In the case of character data (a string is just an array of characters) one such encoding is ASCII (American Standard Code for Information Interchange). However, this is just a convention. There are others (EBCDIC, Unicode…).



Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(	72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29	)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-_	63	3F	?	95	5F	_	127	7F	DEL

Seeq Server

https://sunypoly.seeq.tech/login



Accelerate Industrial Process Analytics

Directory

Seeq

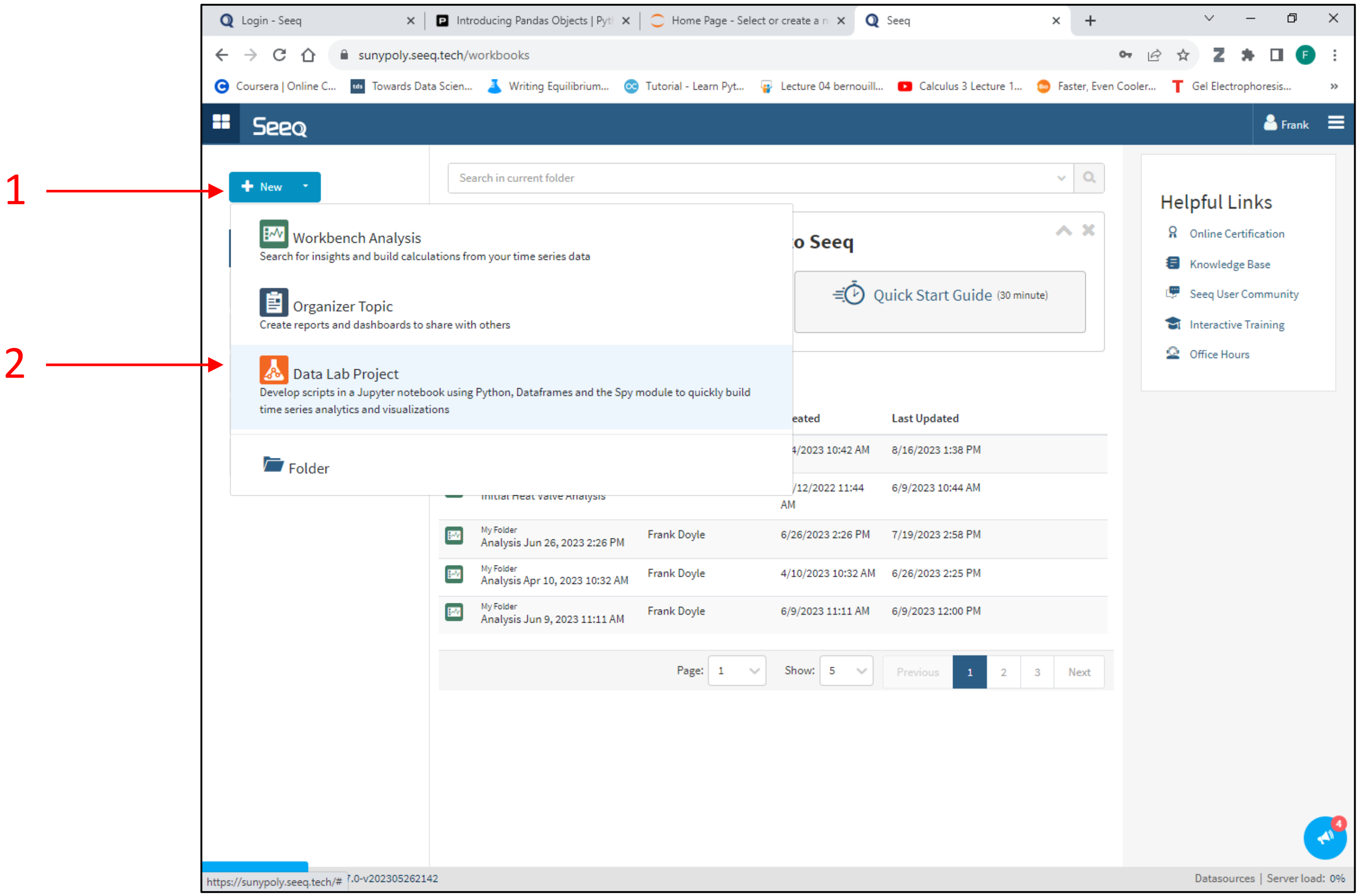
Username:

doylef@sunypoly.edu

Password

•••••••••••••••

Seeq Home Page



Workbench Analysis
Search for insights and build calculations from your time series data

Organizer Topic
Create reports and dashboards to share with others

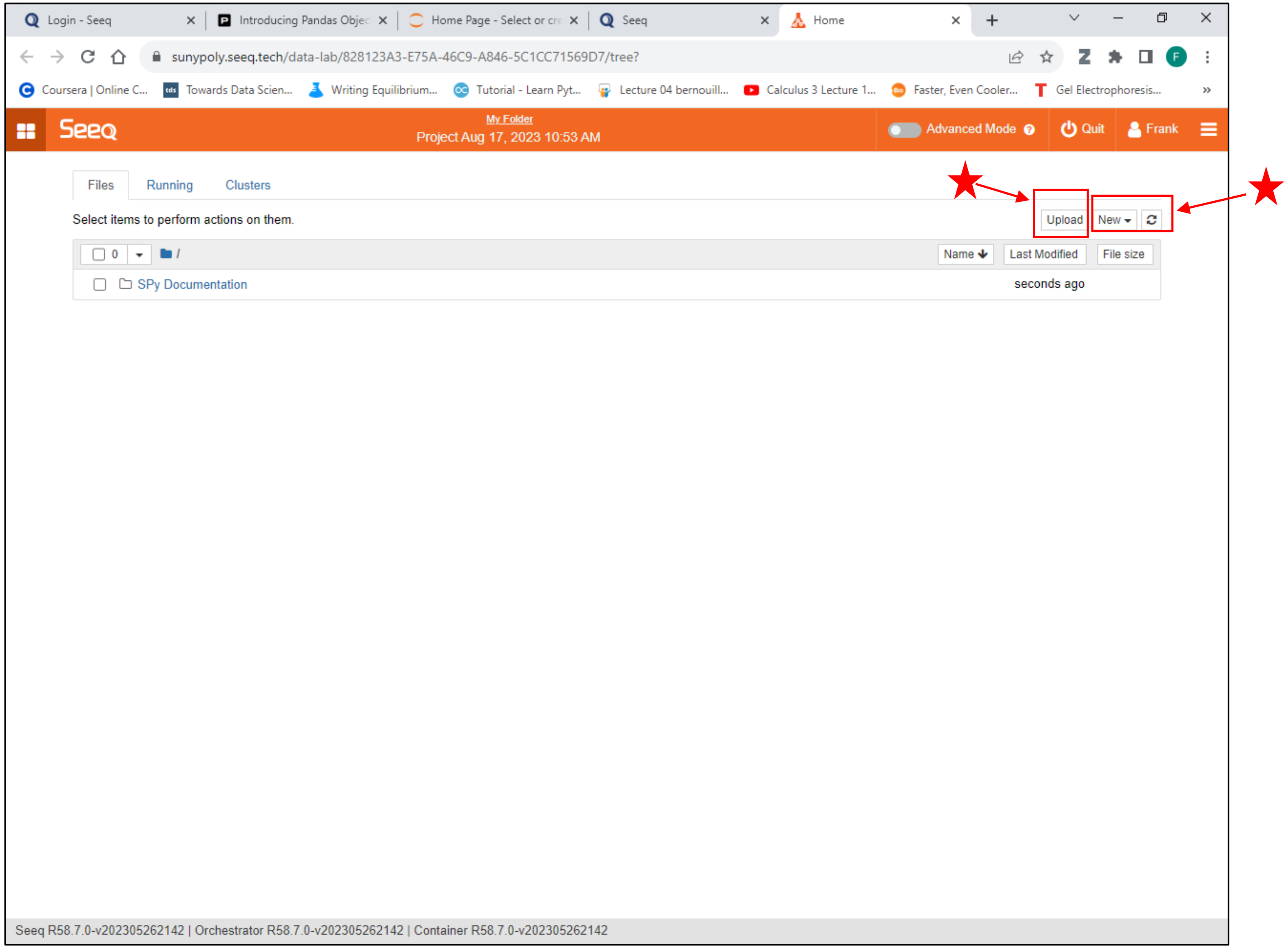
Data Lab Project
Develop scripts in a Jupyter notebook using Python, Dataframes and the Spy module to quickly build time series analytics and visualizations

Folder

		eated	Last Updated
		4/2023 10:42 AM	8/16/2023 1:38 PM
Initial Heat Valve Analysis		/12/2022 11:44 AM	6/9/2023 10:44 AM
My Folder Analysis Jun 26, 2023 2:26 PM	Frank Doyle	6/26/2023 2:26 PM	7/19/2023 2:58 PM
My Folder Analysis Apr 10, 2023 10:32 AM	Frank Doyle	4/10/2023 10:32 AM	6/26/2023 2:25 PM
My Folder Analysis Jun 9, 2023 11:11 AM	Frank Doyle	6/9/2023 11:11 AM	6/9/2023 12:00 PM

Page: 1 Show: 5 Previous 1 2 3 Next

Seeq Data Lab Home

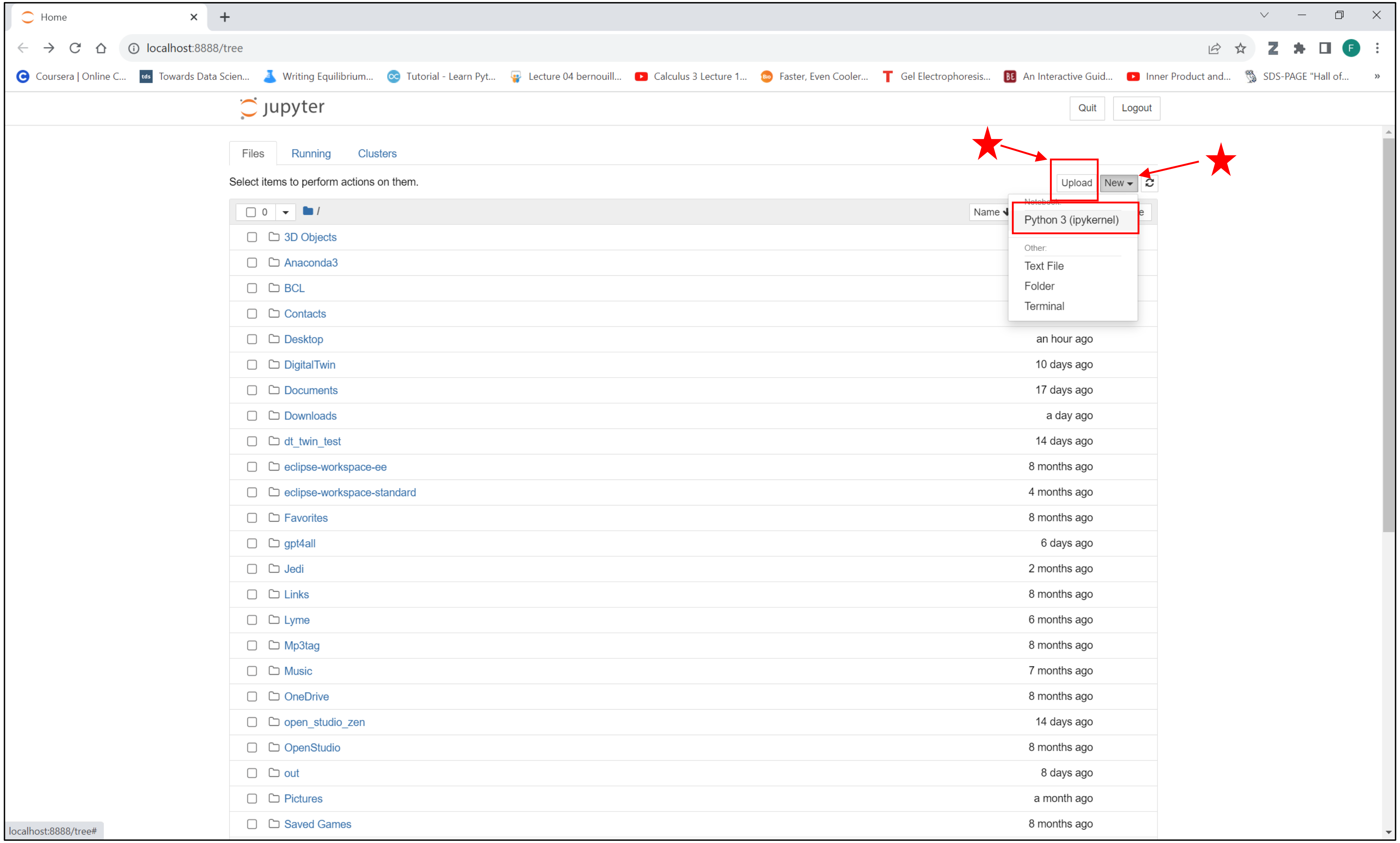


Select items to perform actions on them.

□ 0 /	Name	Last Modified	File size
□ SPy Documentation		seconds ago	

Seeq R58.7.0-v202305262142 | Orchestrator R58.7.0-v202305262142 | Container R58.7.0-v202305262142

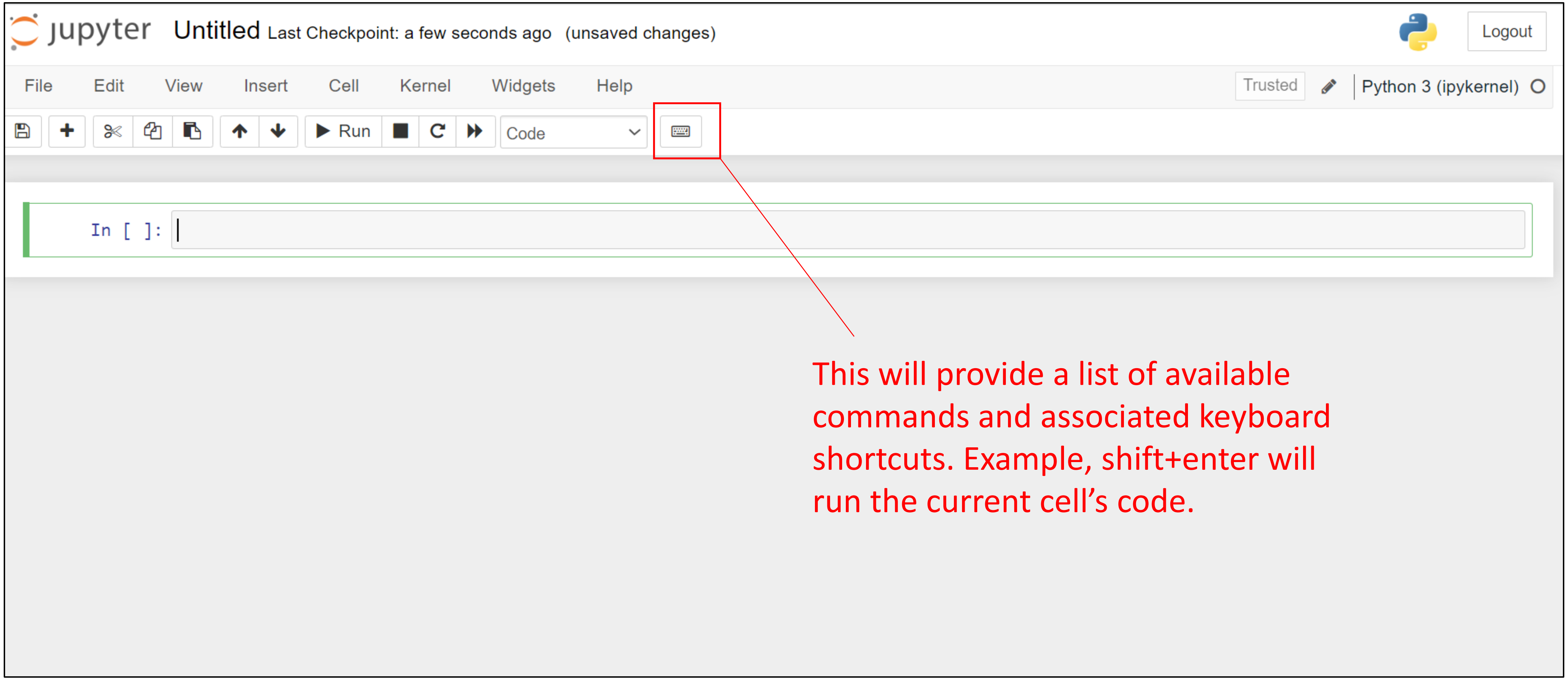
Jupyter Environment



Select items to perform actions on them.

Name	
3D Objects	
Anaconda3	
BCL	
Contacts	
Desktop	an hour ago
DigitalTwin	10 days ago
Documents	17 days ago
Downloads	a day ago
dt_twin_test	14 days ago
eclipse-workspace-ee	8 months ago
eclipse-workspace-standard	4 months ago
Favorites	8 months ago
gpt4all	6 days ago
Jedi	2 months ago
Links	8 months ago
Lyme	6 months ago
Mp3tag	8 months ago
Music	7 months ago
OneDrive	8 months ago
open_studio_zen	14 days ago
OpenStudio	8 months ago
out	8 days ago
Pictures	a month ago
Saved Games	8 months ago

A new, empty notebook…

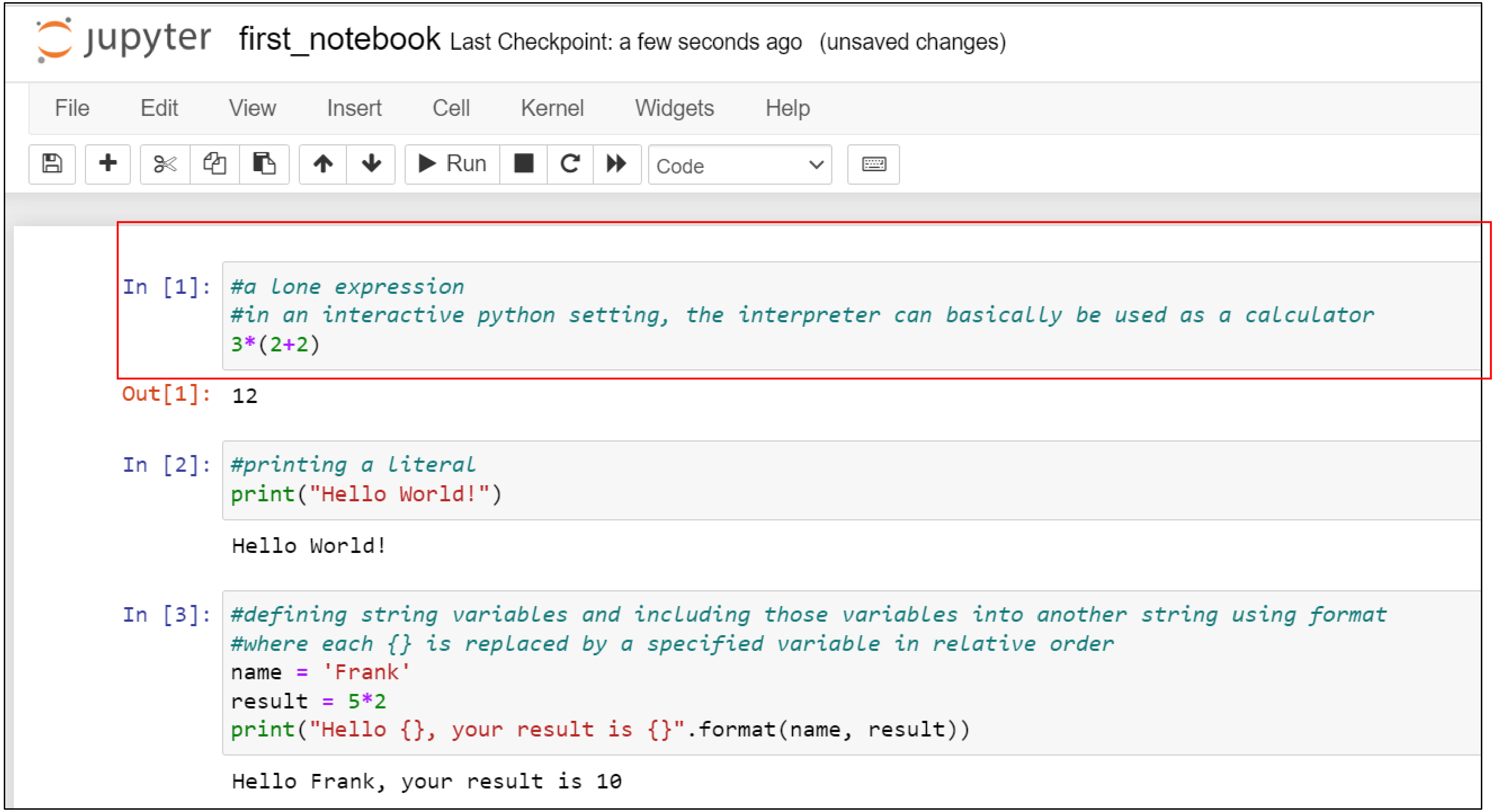


This will provide a list of available commands and associated keyboard shortcuts. Example, shift+enter will run the current cell’s code.

Let’s Start Looking at Code

- Any useful program, regardless of how complicated or simple, typically does a few things:
  - Accepts some form of input (may be from file, interactive prompting or just “hard coded” into the program)
  - Assigns that input to appropriate variables
  - Makes calculations and possibly decisions about program flow based on input values or the result of calculations leading to other calculations
  - Produces output to convey results

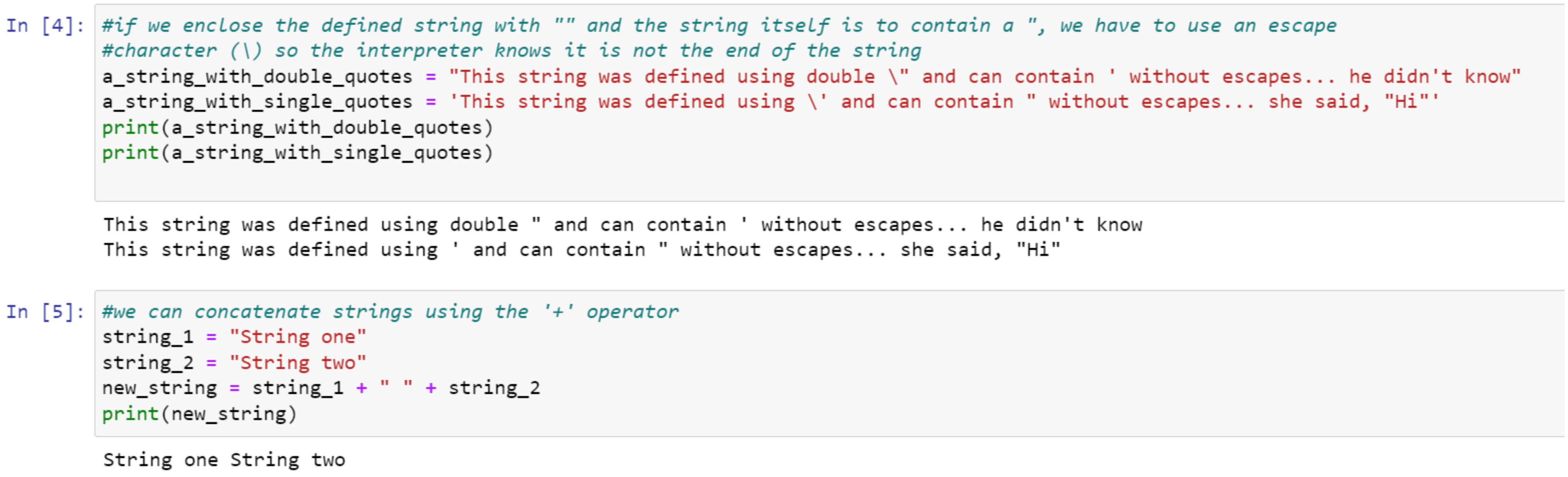
Python Strings and Output


jupyter
first_notebook Last Checkpoint: a few seconds ago (unsaved changes)
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Run
Code

```
In [1]: #a lone expression
        #in an interactive python setting, the interpreter can basically be used as a calculator
        3*(2+2)
Out[1]: 12

In [2]: #printing a literal
        print("Hello World!")

        Hello World!

In [3]: #defining string variables and including those variables into another string using format
        #where each {} is replaced by a specified variable in relative order
        name = 'Frank'
        result = 5*2
        print("Hello {}, your result is {}".format(name, result))

        Hello Frank, your result is 10
```

A “cell”

Python Strings and Output - continued



In [4]:

```
#if we enclose the defined string with "" and the string itself is to contain a ", we have to use an escape
#character (\) so the interpreter knows it is not the end of the string
a_string_with_double_quotes = "This string was defined using double \" and can contain ' without escapes... he didn't know"
a_string_with_single_quotes = 'This string was defined using \' and can contain " without escapes... she said, "Hi"'
print(a_string_with_double_quotes)
print(a_string_with_single_quotes)
```

```
This string was defined using double " and can contain ' without escapes... he didn't know
This string was defined using ' and can contain " without escapes... she said, "Hi"
```

In [5]:

```
#we can concatenate strings using the '+' operator
string_1 = "String one"
string_2 = "String two"
new_string = string_1 + " " + string_2
print(new_string)
```

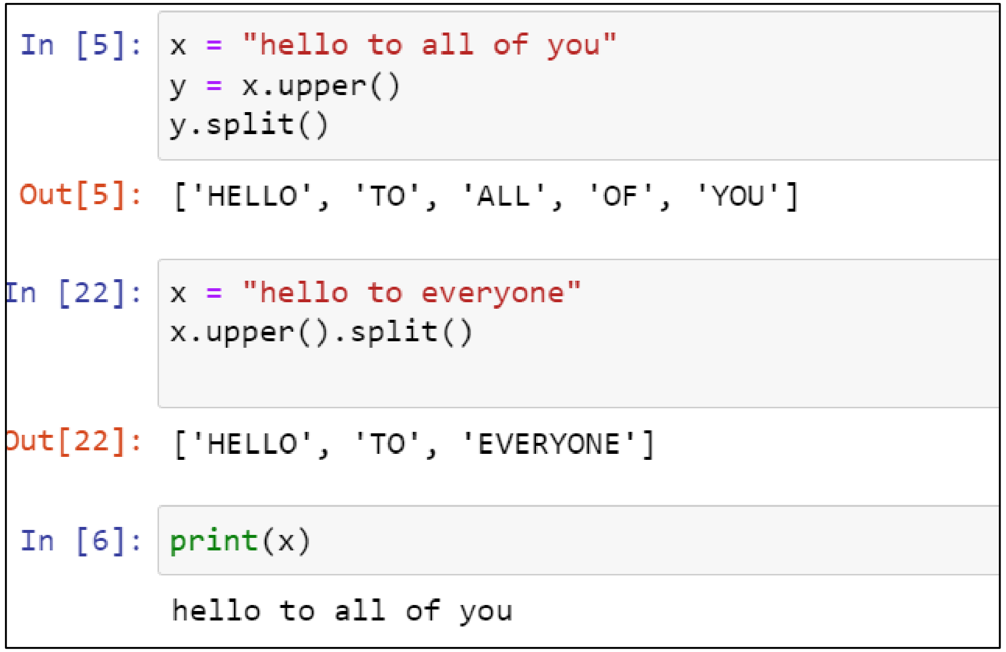
```
String one String two
```

Python Syntax – dot notation

In Python (and other languages), we use dot (".") notation to reference components inside a specified container such as:

- Attributes – values
- Methods (functions) – executable logic

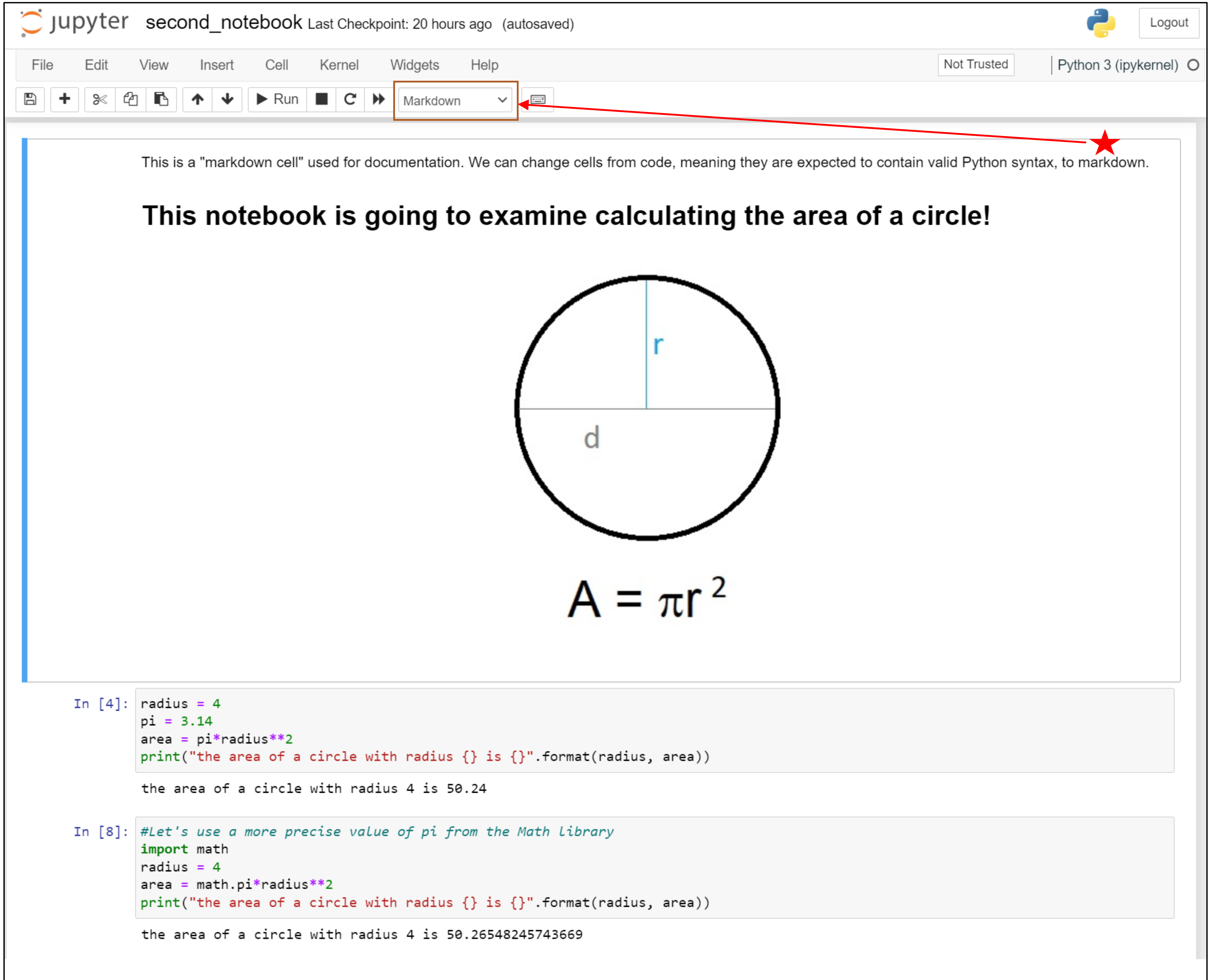
The "container" might be a library, package or module or an object.
A line of code may have multiple dots as the return types of functions may be objects whose attributes or methods are then called.



```
In [5]:  x = "hello to all of you"
         y = x.upper()
         y.split()

Out[5]:  ['HELLO', 'TO', 'ALL', 'OF', 'YOU']

In [22]: x = "hello to everyone"
         x.upper().split()

Out[22]: ['HELLO', 'TO', 'EVERYONE']

In [6]:  print(x)

         hello to all of you
```

To create a cell for documentation purposes, we need to change it’s type from “code” to “markdown”. We can enter text and format it via HTML tags (next slide) and then change from the “edit” mode to “display” by pressing **Run**, or using the keyboard shortcut *Shift+Enter* (which can also be used to execute code cell logic). To re-enter edit mode, you can double click in the cell

The two code cells show the same calculation, first performed with a user defiend variable for the value of pi, then with a high precision value retrieved from the “math” module. Note the “dot” notation used to access the pi component within “math”

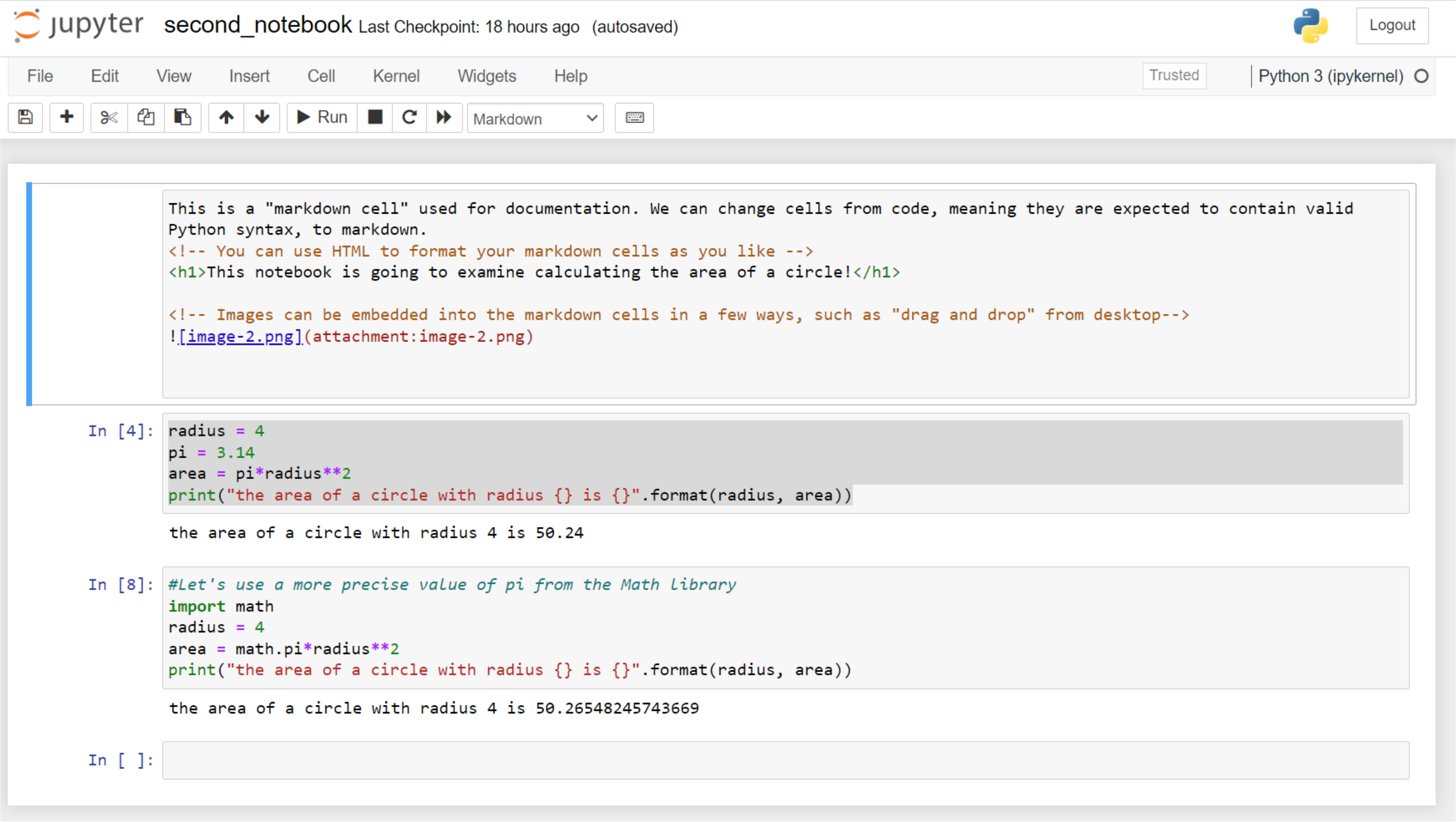


This is a "markdown cell" used for documentation. We can change cells from code, meaning they are expected to contain valid Python syntax, to markdown.

This notebook is going to examine calculating the area of a circle!

$$A = \pi r^2$$

```
In [4]: radius = 4
        pi = 3.14
        area = pi*radius**2
        print("the area of a circle with radius {} is {}".format(radius, area))

        the area of a circle with radius 4 is 50.24

In [8]: #Let's use a more precise value of pi from the Math Library
        import math
        radius = 4
        area = math.pi*radius**2
        print("the area of a circle with radius {} is {}".format(radius, area))

        the area of a circle with radius 4 is 50.26548245743669
```



```
This is a "markdown cell" used for documentation. We can change cells from code, meaning they are expected to contain valid
Python syntax, to markdown.
<!-- You can use HTML to format your markdown cells as you like -->
<h1>This notebook is going to examine calculating the area of a circle!</h1>

<!-- Images can be embedded into the markdown cells in a few ways, such as "drag and drop" from desktop-->
![image-2.png](attachment:image-2.png)
```

In [4]:

```
radius = 4
pi = 3.14
area = pi*radius**2
print("the area of a circle with radius {} is {}".format(radius, area))
```

```
the area of a circle with radius 4 is 50.24
```

In [8]:

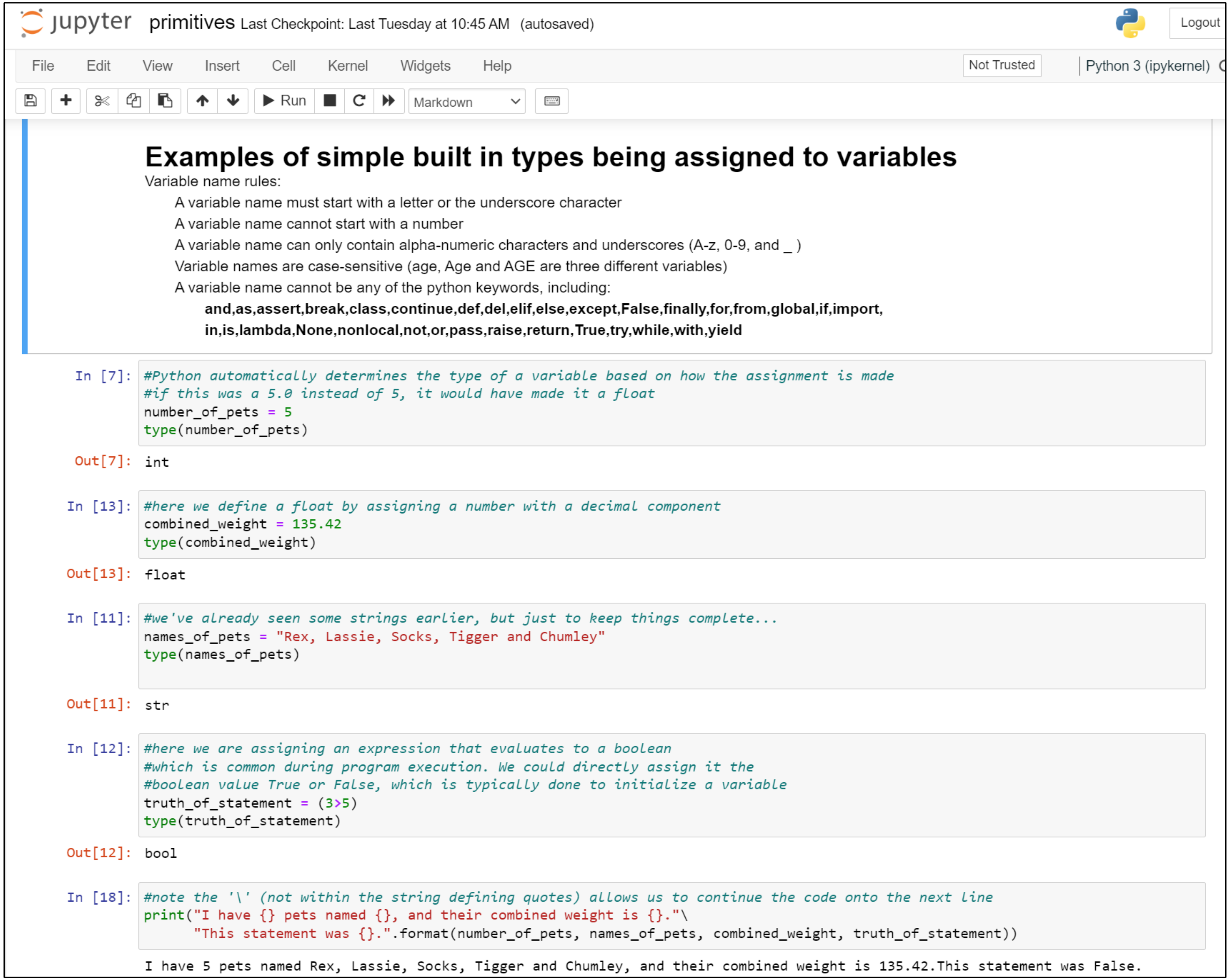
```
#Let's use a more precise value of pi from the Math library
import math
radius = 4
area = math.pi*radius**2
print("the area of a circle with radius {} is {}".format(radius, area))
```

```
the area of a circle with radius 4 is 50.26548245743669
```

In [ ]:

Python Syntax – arithmetic operations

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus/Remainder	x % y
**	Exponentiation	x ** y
//	Floor division	x // y



Examples of simple built in types being assigned to variables

Variable name rules:

A variable name must start with a letter or the underscore character

A variable name cannot start with a number

A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)

Variable names are case-sensitive (age, Age and AGE are three different variables)

A variable name cannot be any of the python keywords, including:

and,as,assert,break,class,continue,def,del,elif,else,except,False,finally,for,from,global,if,import, in,is,lambda,None,nonlocal,not,or,pass,raise,return,True,try,while,with,yield

In [7]:
```
#Python automatically determines the type of a variable based on how the assignment is made
#if this was a 5.0 instead of 5, it would have made it a float
number_of_pets = 5
type(number_of_pets)
```

Out[7]:
```
int
```

In [13]:
```
#here we define a float by assigning a number with a decimal component
combined_weight = 135.42
type(combined_weight)
```

Out[13]:
```
float
```

In [11]:
```
#we've already seen some strings earlier, but just to keep things complete...
names_of_pets = "Rex, Lassie, Socks, Tigger and Chumley"
type(names_of_pets)
```

Out[11]:
```
str
```

In [12]:
```
#here we are assigning an expression that evaluates to a boolean
#which is common during program execution. We could directly assign it the
#boolean value True or False, which is typically done to initialize a variable
truth_of_statement = (3>5)
type(truth_of_statement)
```

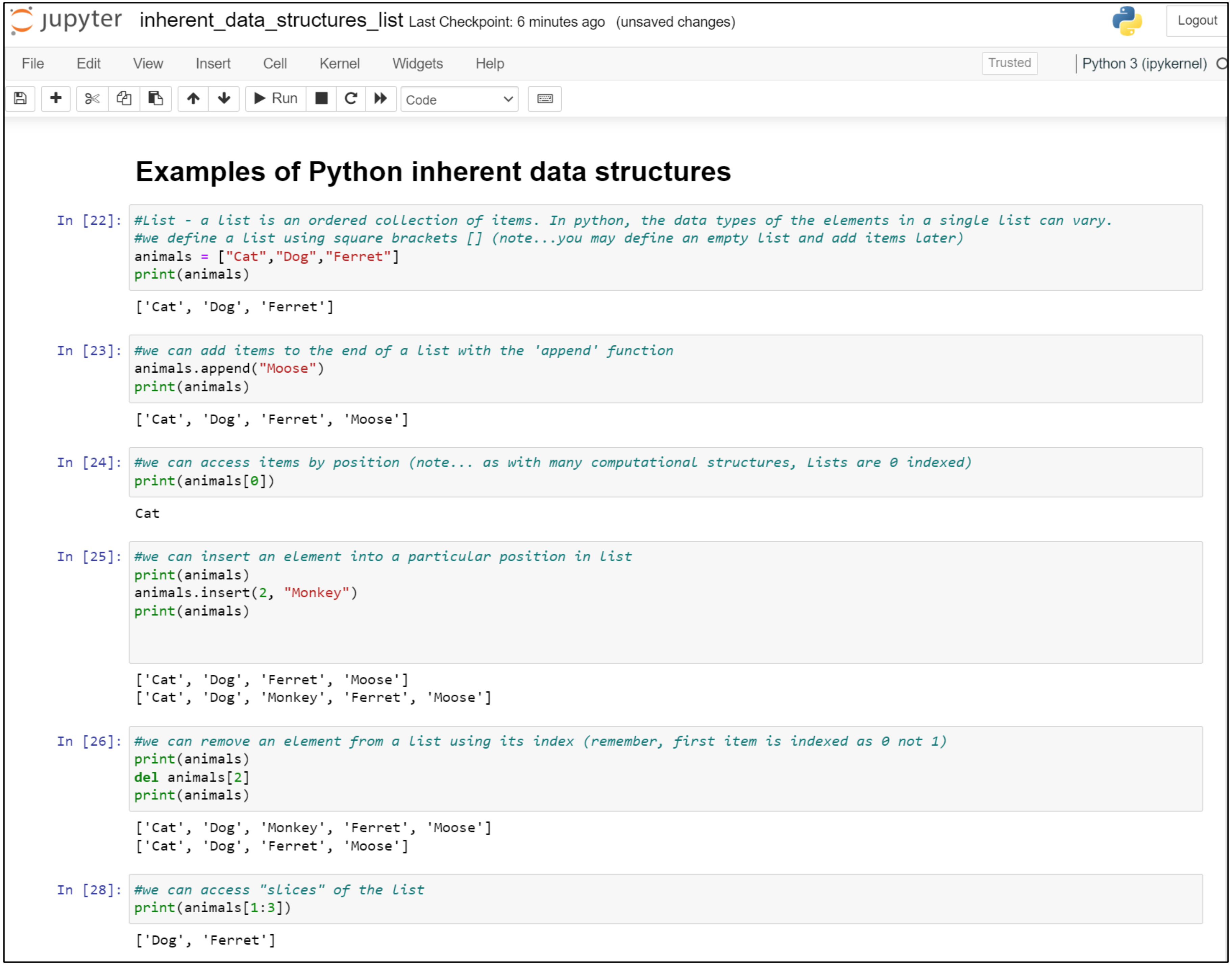
Out[12]:
```
bool
```

In [18]:
```
#note the '\' (not within the string defining quotes) allows us to continue the code onto the next line
print("I have {} pets named {}, and their combined weight is {}."\
      "This statement was {}.".format(number_of_pets, names_of_pets, combined_weight, truth_of_statement))
```

```
I have 5 pets named Rex, Lassie, Socks, Tigger and Chumley, and their combined weight is 135.42.This statement was False.
```

Boolean Expressions (aka Conditional tests)

Operator	Meaning	Example
>	Greater than – True if left operand is greater than right operand	a > b
<	Less than - True if left operand is less than right operand	a < b
==	Equal to – True if both operands are equal	a == b
!=	Not equal to – True operands are not equal	a != b
>=	Greater than or equal to – True if left operand is greater or equal to right operand	a >= b
<=	Less than or equal to – True if left operand is less than or equal to right operand	a <= b
and	True if both left and right expressions are True	exp1 **and** exp2
or	True if either left or right expressions are true	exp1 **or** exp2
not	True if following expression is False	**not** exp1



Examples of Python inherent data structures

In [22]:
```
#List - a list is an ordered collection of items. In python, the data types of the elements in a single list can vary.
#we define a list using square brackets [] (note...you may define an empty list and add items later)
animals = ["Cat","Dog","Ferret"]
print(animals)
```

```
['Cat', 'Dog', 'Ferret']
```

In [23]:
```
#we can add items to the end of a list with the 'append' function
animals.append("Moose")
print(animals)
```

```
['Cat', 'Dog', 'Ferret', 'Moose']
```

In [24]:
```
#we can access items by position (note... as with many computational structures, Lists are 0 indexed)
print(animals[0])
```

```
Cat
```

In [25]:
```
#we can insert an element into a particular position in list
print(animals)
animals.insert(2, "Monkey")
print(animals)
```

```
['Cat', 'Dog', 'Ferret', 'Moose']
['Cat', 'Dog', 'Monkey', 'Ferret', 'Moose']
```

In [26]:
```
#we can remove an element from a list using its index (remember, first item is indexed as 0 not 1)
print(animals)
del animals[2]
print(animals)
```

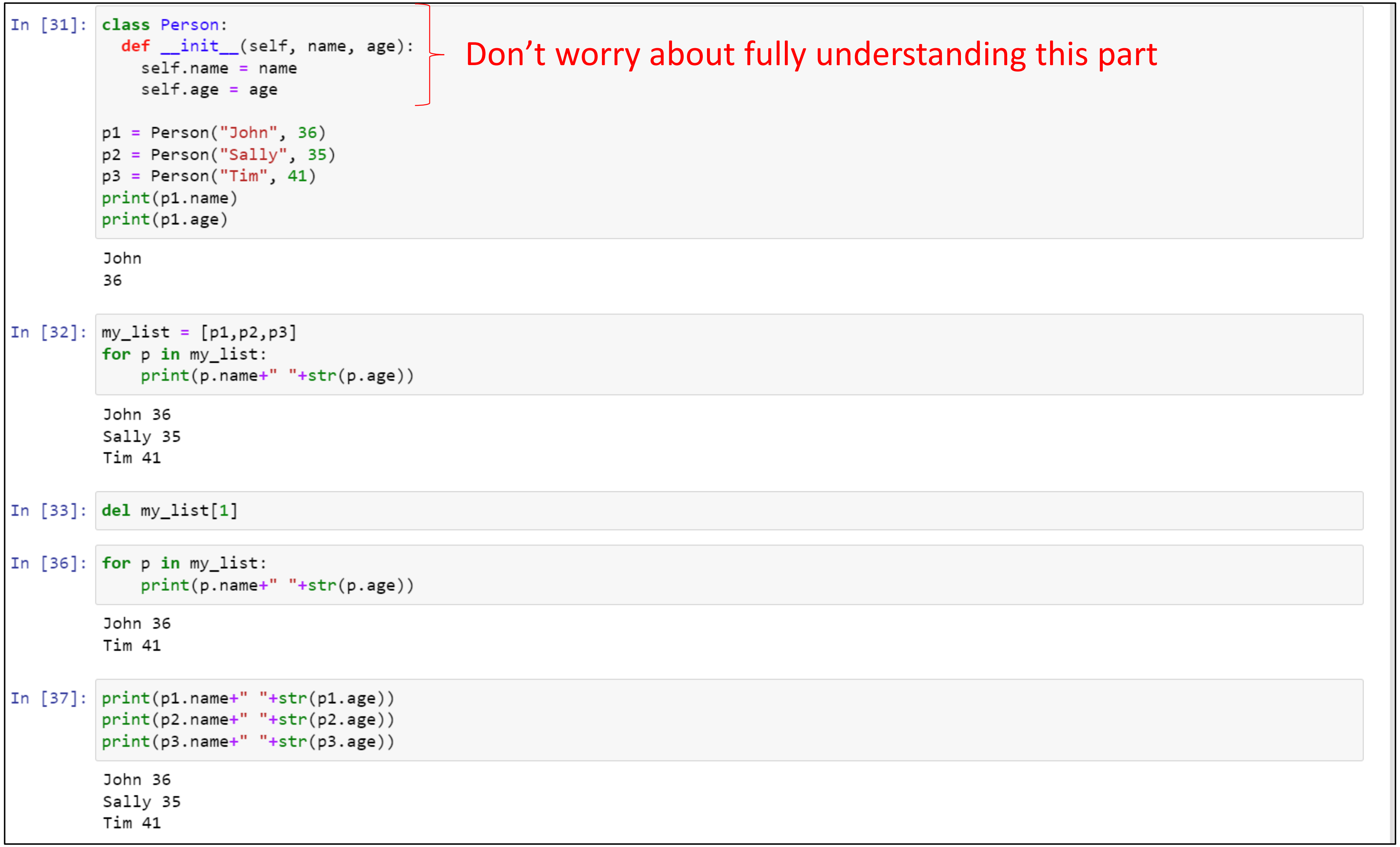
```
['Cat', 'Dog', 'Monkey', 'Ferret', 'Moose']
['Cat', 'Dog', 'Ferret', 'Moose']
```

In [28]:
```
#we can access "slices" of the list
print(animals[1:3])
```

```
['Dog', 'Ferret']
```

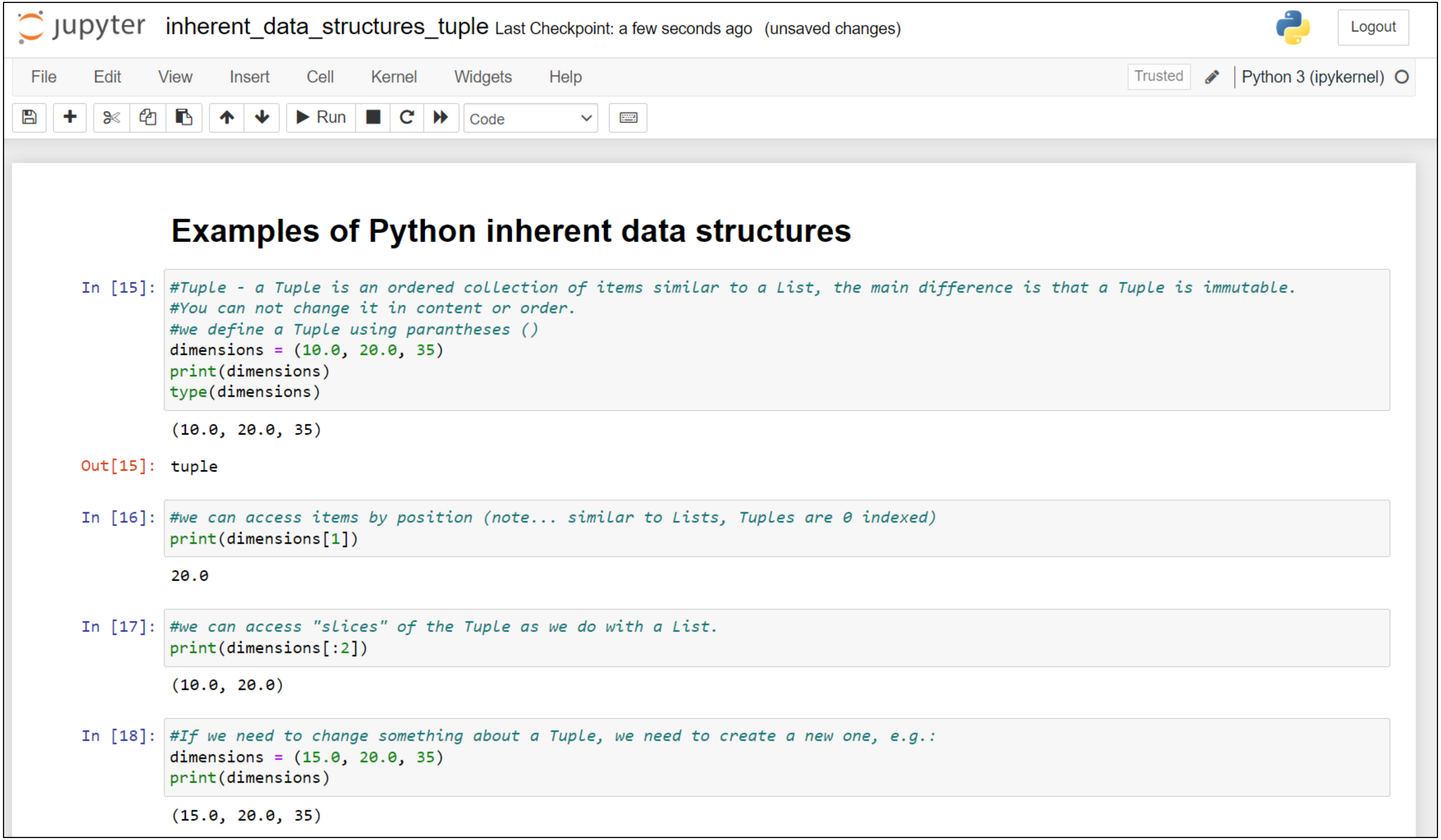
A note on the “del” keyword…

“del” removes a specific reference to an object, it does not actually eradicate that object from existence.



Don’t worry about fully understanding this part

```
In [31]: class Person:
           def __init__(self, name, age):
             self.name = name
             self.age = age

         p1 = Person("John", 36)
         p2 = Person("Sally", 35)
         p3 = Person("Tim", 41)
         print(p1.name)
         print(p1.age)

         John
         36

In [32]: my_list = [p1,p2,p3]
         for p in my_list:
             print(p.name+" "+str(p.age))

         John 36
         Sally 35
         Tim 41

In [33]: del my_list[1]

In [36]: for p in my_list:
             print(p.name+" "+str(p.age))

         John 36
         Tim 41

In [37]: print(p1.name+" "+str(p1.age))
         print(p2.name+" "+str(p2.age))
         print(p3.name+" "+str(p3.age))

         John 36
         Sally 35
         Tim 41
```



Examples of Python inherent data structures

In [15]:

```
#Tuple - a Tuple is an ordered collection of items similar to a List, the main difference is that a Tuple is immutable.
#You can not change it in content or order.
#we define a Tuple using parantheses ()
dimensions = (10.0, 20.0, 35)
print(dimensions)
type(dimensions)
```

```
(10.0, 20.0, 35)
```

Out[15]:

```
tuple
```

In [16]:

```
#we can access items by position (note... similar to Lists, Tuples are 0 indexed)
print(dimensions[1])
```

```
20.0
```

In [17]:

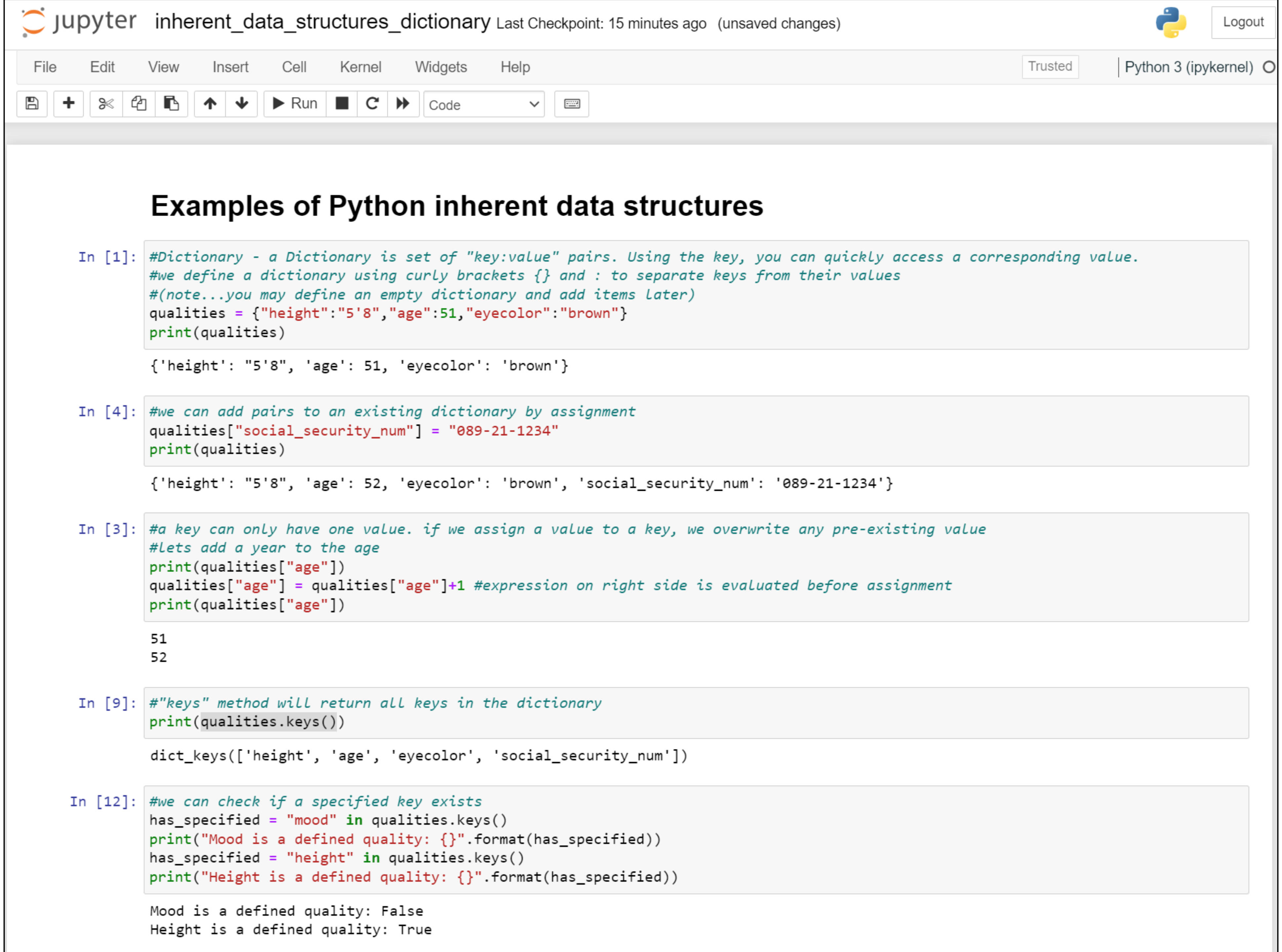
```
#we can access "slices" of the Tuple as we do with a List.
print(dimensions[:2])
```

```
(10.0, 20.0)
```

In [18]:

```
#If we need to change something about a Tuple, we need to create a new one, e.g.:
dimensions = (15.0, 20.0, 35)
print(dimensions)
```

```
(15.0, 20.0, 35)
```



Examples of Python inherent data structures

In [1]:

```
#Dictionary - a Dictionary is set of "key:value" pairs. Using the key, you can quickly access a corresponding value.
#we define a dictionary using curly brackets {} and : to separate keys from their values
#(note...you may define an empty dictionary and add items later)
qualities = {"height":"5'8","age":51,"eyecolor":"brown"}
print(qualities)
```

```
{'height': "5'8", 'age': 51, 'eyecolor': 'brown'}
```

In [4]:

```
#we can add pairs to an existing dictionary by assignment
qualities["social_security_num"] = "089-21-1234"
print(qualities)
```

```
{'height': "5'8", 'age': 52, 'eyecolor': 'brown', 'social_security_num': '089-21-1234'}
```

In [3]:

```
#a key can only have one value. if we assign a value to a key, we overwrite any pre-existing value
#lets add a year to the age
print(qualities["age"])
qualities["age"] = qualities["age"]+1 #expression on right side is evaluated before assignment
print(qualities["age"])
```

```
51
52
```

In [9]:

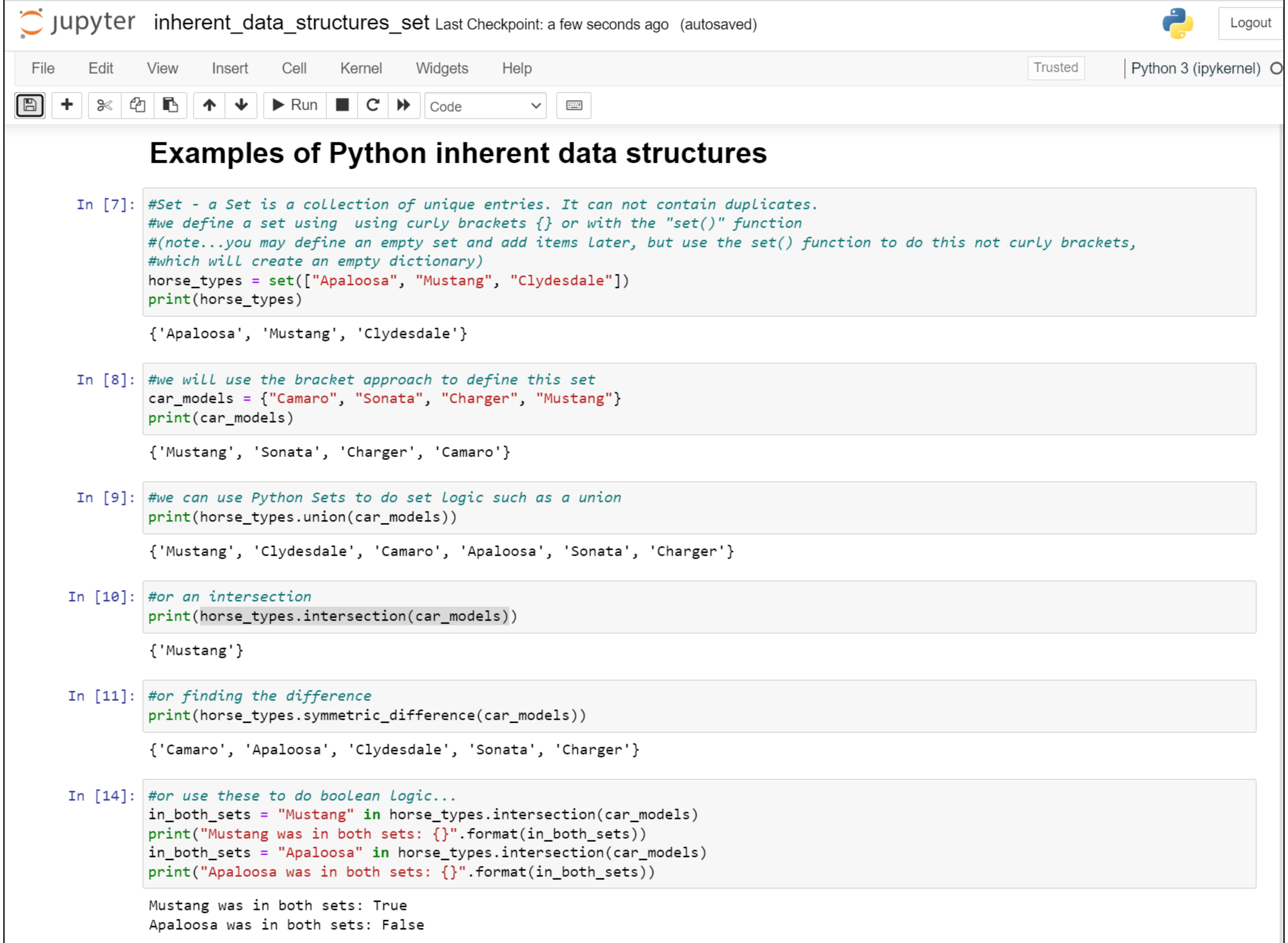
```
#"keys" method will return all keys in the dictionary
print(qualities.keys())
```

```
dict_keys(['height', 'age', 'eyecolor', 'social_security_num'])
```

In [12]:

```
#we can check if a specified key exists
has_specified = "mood" in qualities.keys()
print("Mood is a defined quality: {}".format(has_specified))
has_specified = "height" in qualities.keys()
print("Height is a defined quality: {}".format(has_specified))
```

```
Mood is a defined quality: False
Height is a defined quality: True
```



Examples of Python inherent data structures

In [7]:

```
#Set - a Set is a collection of unique entries. It can not contain duplicates.
#we define a set using  using curly brackets {} or with the "set()" function
#(note...you may define an empty set and add items later, but use the set() function to do this not curly brackets,
#which will create an empty dictionary)
horse_types = set(["Apaloosa", "Mustang", "Clydesdale"])
print(horse_types)
```

```
{'Apaloosa', 'Mustang', 'Clydesdale'}
```

In [8]:

```
#we will use the bracket approach to define this set
car_models = {"Camaro", "Sonata", "Charger", "Mustang"}
print(car_models)
```

```
{'Mustang', 'Sonata', 'Charger', 'Camaro'}
```

In [9]:

```
#we can use Python Sets to do set logic such as a union
print(horse_types.union(car_models))
```

```
{'Mustang', 'Clydesdale', 'Camaro', 'Apaloosa', 'Sonata', 'Charger'}
```

In [10]:

```
#or an intersection
print(horse_types.intersection(car_models))
```

```
{'Mustang'}
```

In [11]:

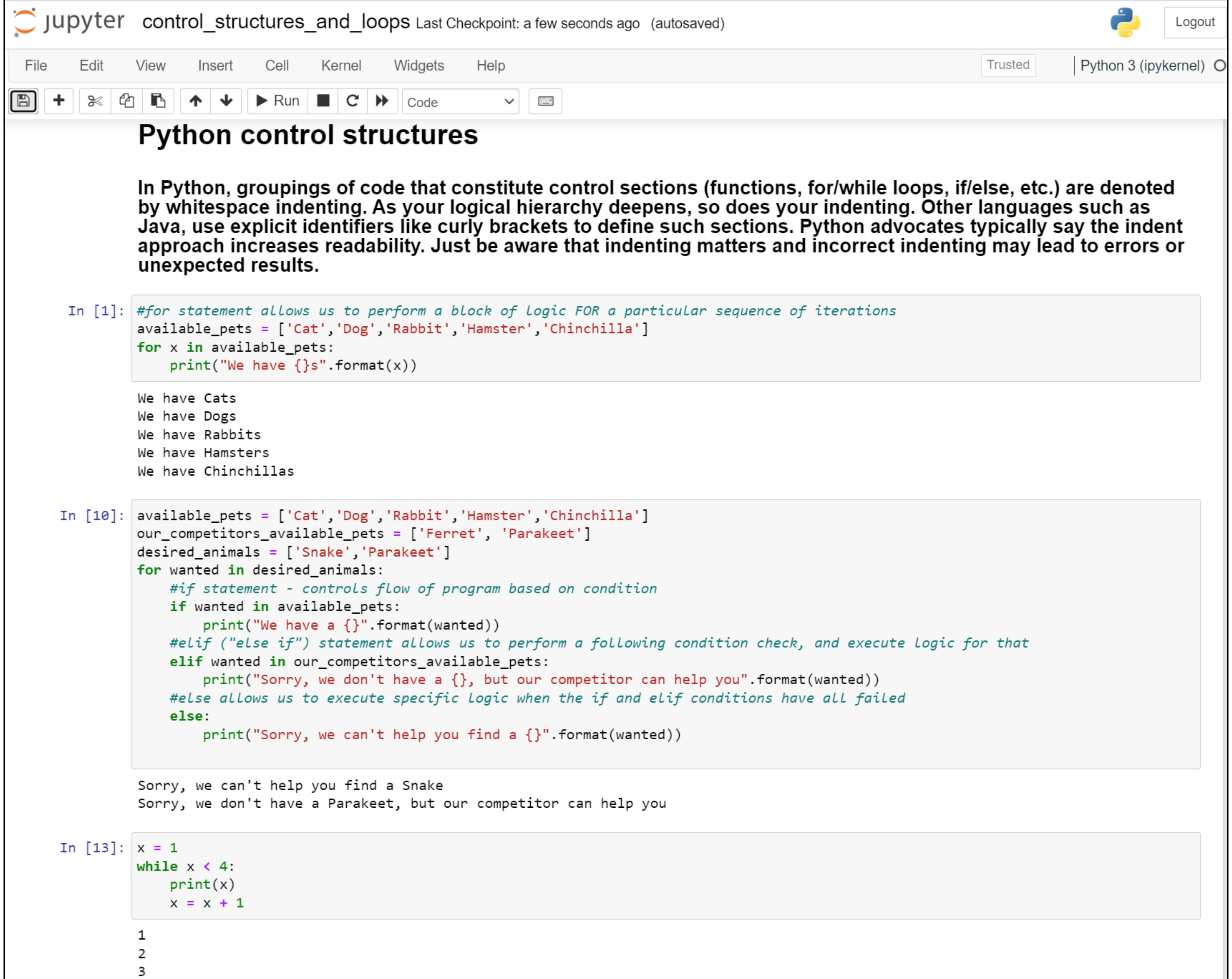
```
#or finding the difference
print(horse_types.symmetric_difference(car_models))
```

```
{'Camaro', 'Apaloosa', 'Clydesdale', 'Sonata', 'Charger'}
```

In [14]:

```
#or use these to do boolean logic...
in_both_sets = "Mustang" in horse_types.intersection(car_models)
print("Mustang was in both sets: {}".format(in_both_sets))
in_both_sets = "Apaloosa" in horse_types.intersection(car_models)
print("Apaloosa was in both sets: {}".format(in_both_sets))
```

```
Mustang was in both sets: True
Apaloosa was in both sets: False
```



Python control structures

In Python, groupings of code that constitute control sections (functions, for/while loops, if/else, etc.) are denoted by whitespace indenting. As your logical hierarchy deepens, so does your indenting. Other languages such as Java, use explicit identifiers like curly brackets to define such sections. Python advocates typically say the indent approach increases readability. Just be aware that indenting matters and incorrect indenting may lead to errors or unexpected results.

In [1]:

```
#for statement allows us to perform a block of logic FOR a particular sequence of iterations
available_pets = ['Cat','Dog','Rabbit','Hamster','Chinchilla']
for x in available_pets:
    print("We have {}s".format(x))
```

```
We have Cats
We have Dogs
We have Rabbits
We have Hamsters
We have Chinchillas
```

In [10]:

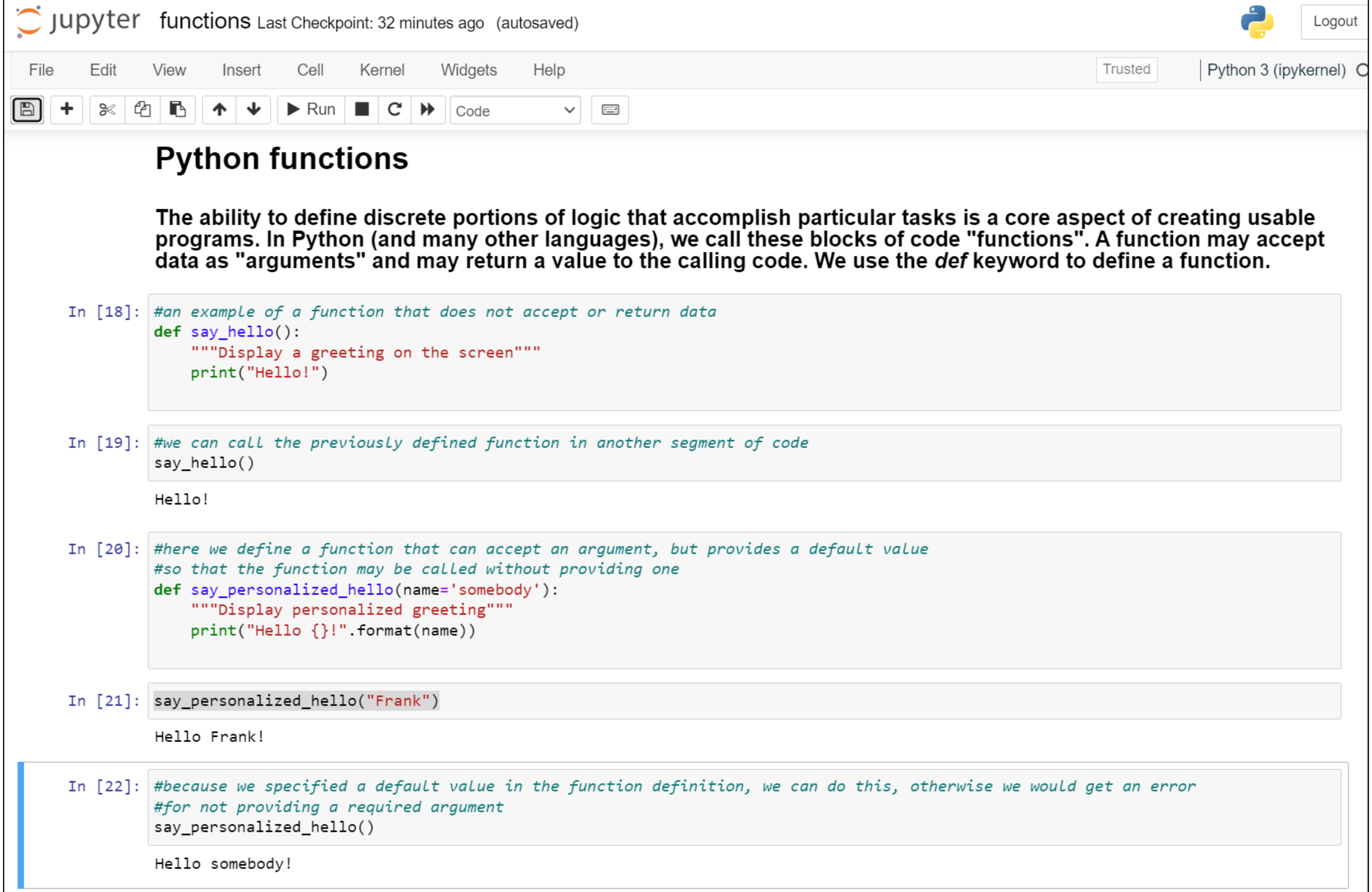
```
available_pets = ['Cat','Dog','Rabbit','Hamster','Chinchilla']
our_competitors_available_pets = ['Ferret', 'Parakeet']
desired_animals = ['Snake','Parakeet']
for wanted in desired_animals:
    #if statement - controls flow of program based on condition
    if wanted in available_pets:
        print("We have a {}".format(wanted))
    #elif ("else if") statement allows us to perform a following condition check, and execute logic for that
    elif wanted in our_competitors_available_pets:
        print("Sorry, we don't have a {}, but our competitor can help you".format(wanted))
    #else allows us to execute specific logic when the if and elif conditions have all failed
    else:
        print("Sorry, we can't help you find a {}".format(wanted))
```

```
Sorry, we can't help you find a Snake
Sorry, we don't have a Parakeet, but our competitor can help you
```

In [13]:

```
x = 1
while x < 4:
    print(x)
    x = x + 1
```

```
1
2
3
```



Python functions

The ability to define discrete portions of logic that accomplish particular tasks is a core aspect of creating usable programs. In Python (and many other languages), we call these blocks of code "functions". A function may accept data as "arguments" and may return a value to the calling code. We use the *def* keyword to define a function.

```
In [18]: #an example of a function that does not accept or return data
         def say_hello():
             """Display a greeting on the screen"""
             print("Hello!")
```

```
In [19]: #we can call the previously defined function in another segment of code
         say_hello()
```

```
Hello!
```

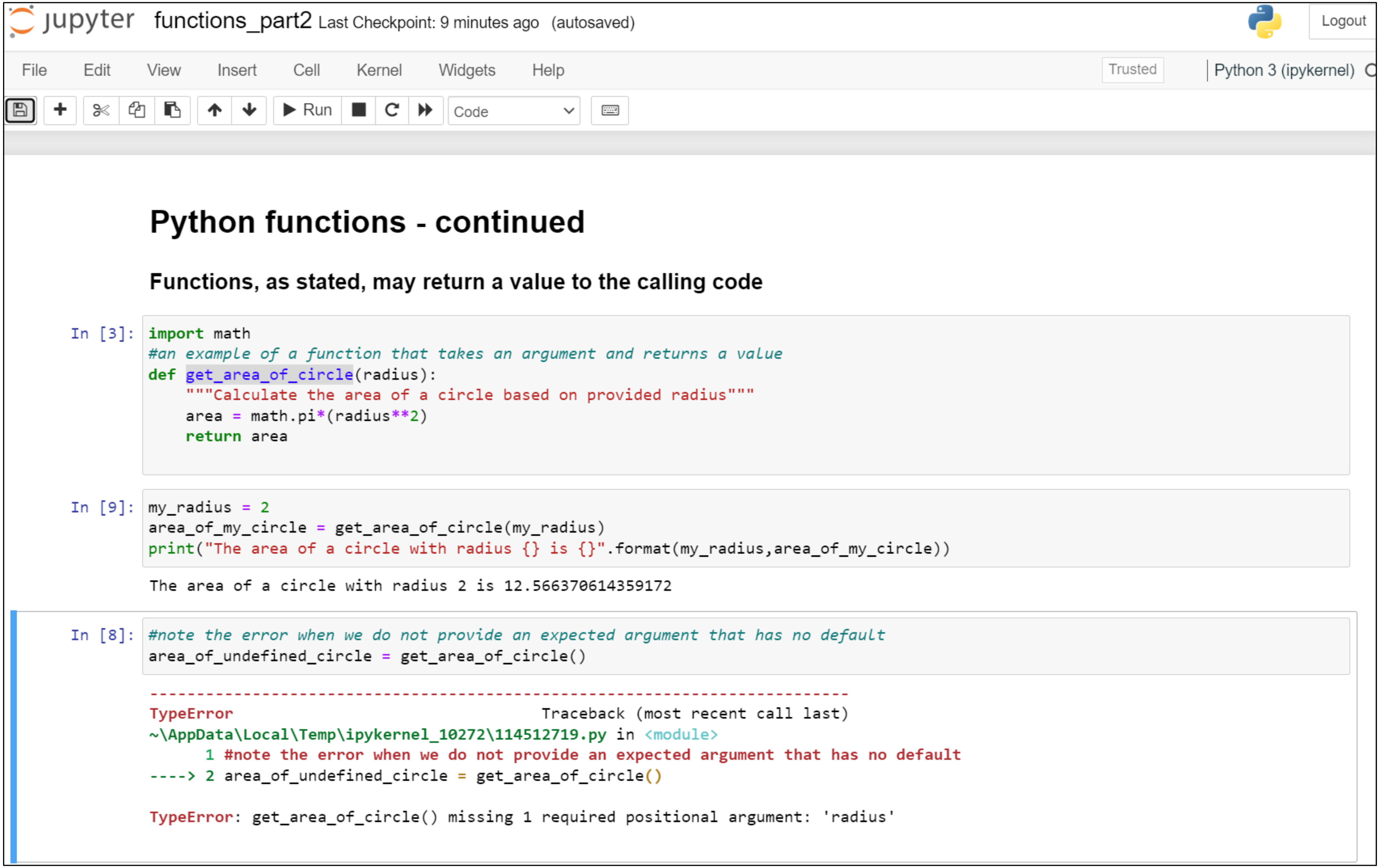
```
In [20]: #here we define a function that can accept an argument, but provides a default value
         #so that the function may be called without providing one
         def say_personalized_hello(name='somebody'):
             """Display personalized greeting"""
             print("Hello {}!".format(name))
```

```
In [21]: say_personalized_hello("Frank")
```

```
Hello Frank!
```

```
In [22]: #because we specified a default value in the function definition, we can do this, otherwise we would get an error
         #for not providing a required argument
         say_personalized_hello()
```

```
Hello somebody!
```



Python functions - continued

Functions, as stated, may return a value to the calling code

In [3]:

```
import math
#an example of a function that takes an argument and returns a value
def get_area_of_circle(radius):
    """Calculate the area of a circle based on provided radius"""
    area = math.pi*(radius**2)
    return area
```

In [9]:

```
my_radius = 2
area_of_my_circle = get_area_of_circle(my_radius)
print("The area of a circle with radius {} is {}".format(my_radius,area_of_my_circle))
```

```
The area of a circle with radius 2 is 12.566370614359172
```

In [8]:

```
#note the error when we do not provide an expected argument that has no default
area_of_undefined_circle = get_area_of_circle()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_10272\114512719.py in <module>
      1 #note the error when we do not provide an expected argument that has no default
----> 2 area_of_undefined_circle = get_area_of_circle()

TypeError: get_area_of_circle() missing 1 required positional argument: 'radius'
```

Homework

- Reading:
  - Textbook – Sections 2.3,3.1
- Activity:
  - Login to Seeq.
  - Create a Data Lab Project.
  - Create a new notebook.
  - Execute some code (even if you just replicate some of the code from these slides). Just prove to yourself that you can use the environment.