Intro to Python

By Craig Sakuma

- WiFi Network: GA Guest
- Password: yellowpencil

Introductions

Craig Sakuma

- Data Science Consultant
- General Assembly Instructor for Data Science
- Co-founder of Deal Décor
- Wharton MBA
- Northwestern University B.Eng

Fun Fact

Developed a novelty BBQ product that was featured in USA Today

Class Introductions

- Name
- What's your job?
- How do you plan to apply skills your Python skills?
- Fun Fact

Objectives for Class

- Build strong foundation of Python
- Remove barriers/frustration
- Enable you to approach more advanced topics
  - Data Munging
  - Data Visualization
  - Machine Learning

HAVE FUN!

Course Structure

- Lectures on topics
  - Interaction is good
  - Feel free to ask questions
  - If there's not enough time to cover questions, we'll put it in a parking lot for after class
- Hands on exercises
  - Pair programming
  - Mix up partners

Schedule

Day	Time	Topic
Day 1	6:30 – 7:00	Overview of Python
	7:00 – 8:30	Python Fundamentals
	8:30 – 9:30	If Statements
Day 2	6:30 – 7:30	Lists, Tuples and Dictionaries
	7:30 – 8:30	For Loops
	8:30 – 9:30	Importing Packages and Datetime

Why Python?

- Readability
- Dynamic typing
- Supports multiple programming paradigms
  - Object oriented
  - Functional
  - Procedural

Libraries of Tools for Data Analysis

What is Anaconda?

- Distribution of Python and commonly used libraries of tools
- Easier than individually installing many libraries
- Ensures the versions of each library are compatible with each other

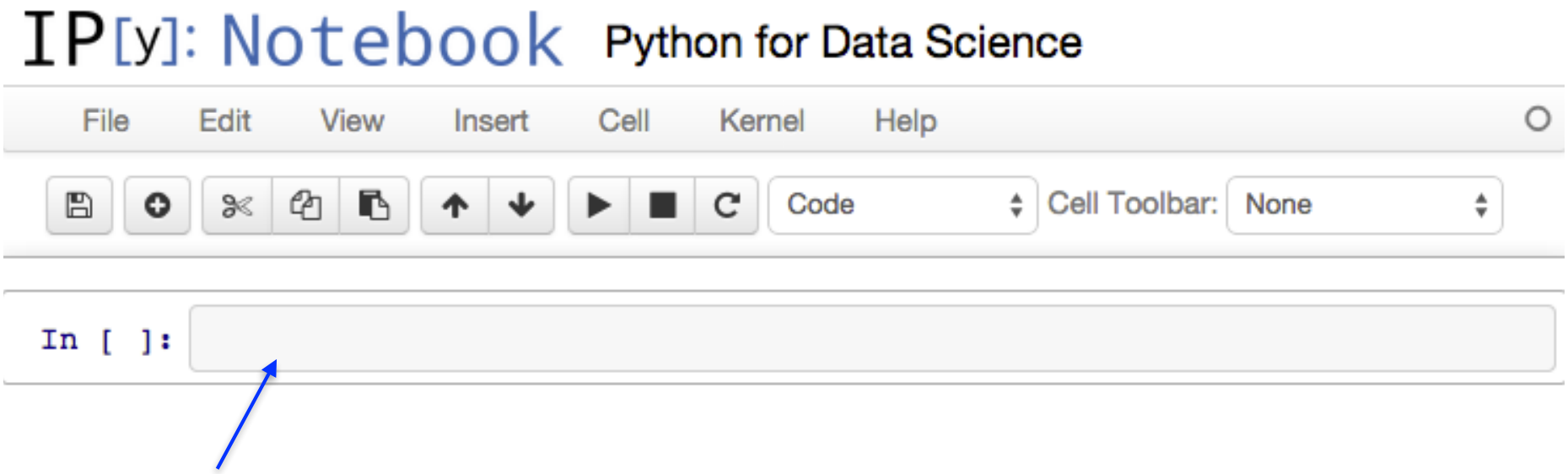
How to Interact with Python

There are several ways to interact with python

- Python Command Line
- Operating System Command Line
- iPython
- iPython Notebook

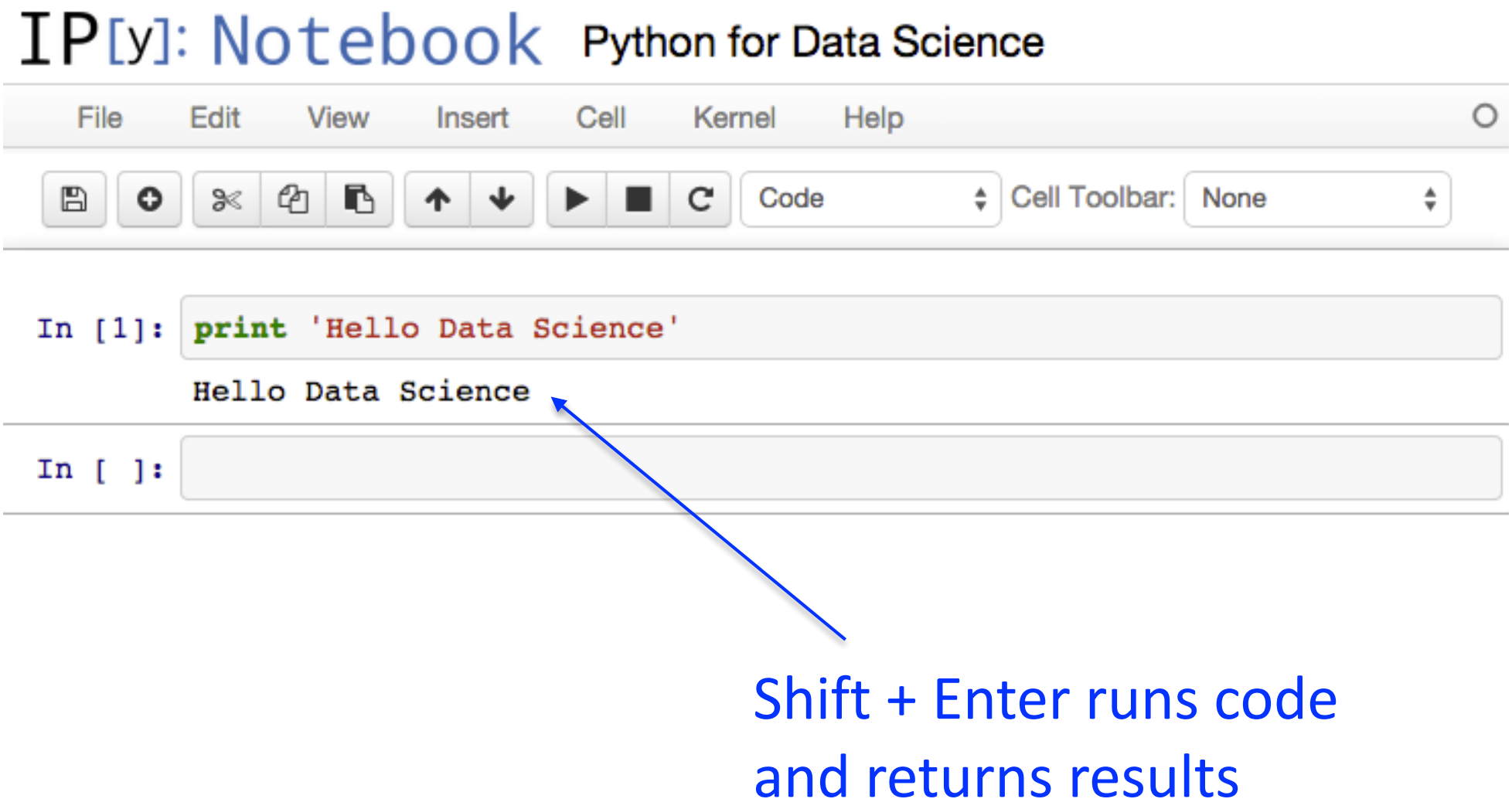
We'll be using iPython Notebooks

iPython Notebook Basics



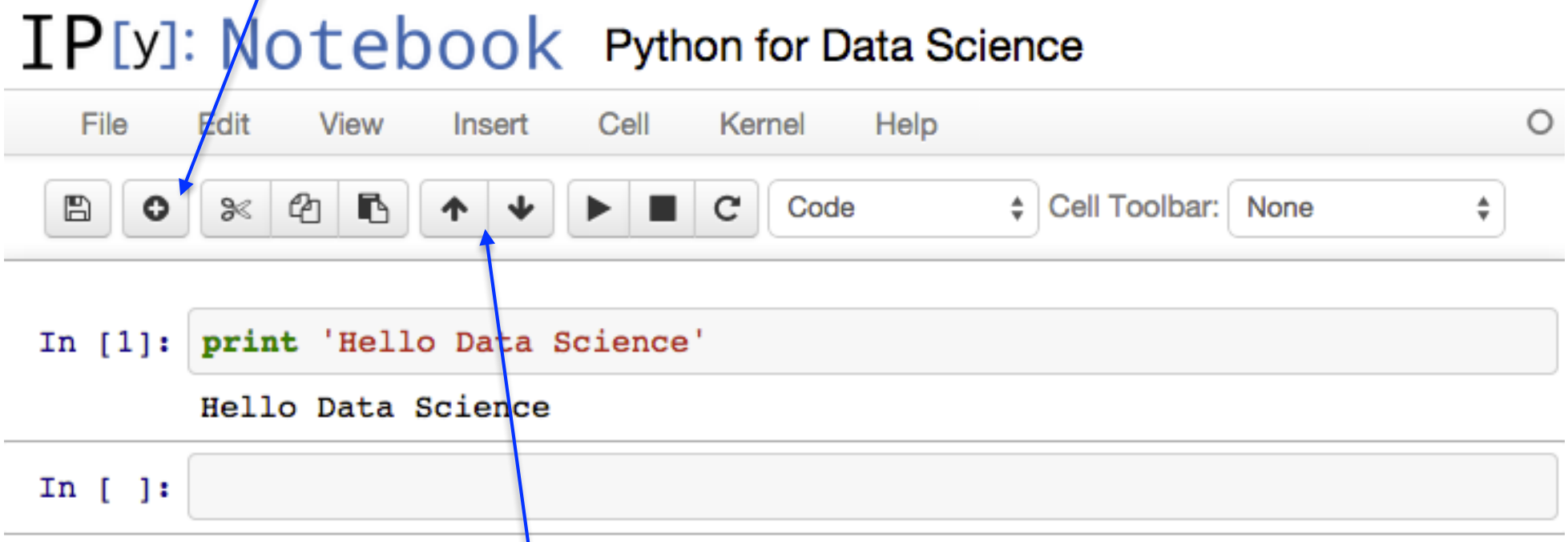
Enter code here

iPython Notebook Basics



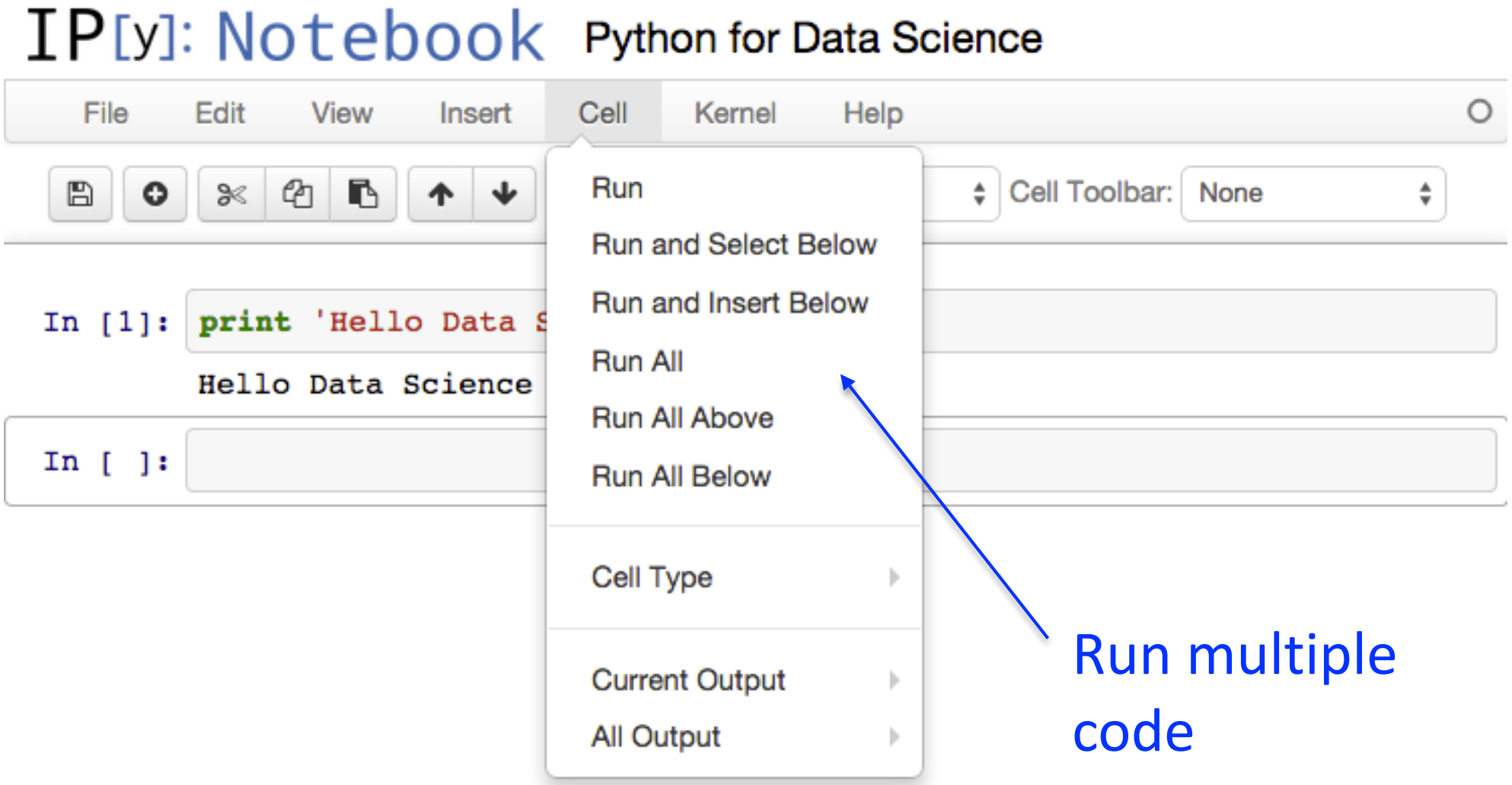
iPython Notebook Basics

Add more code blocks

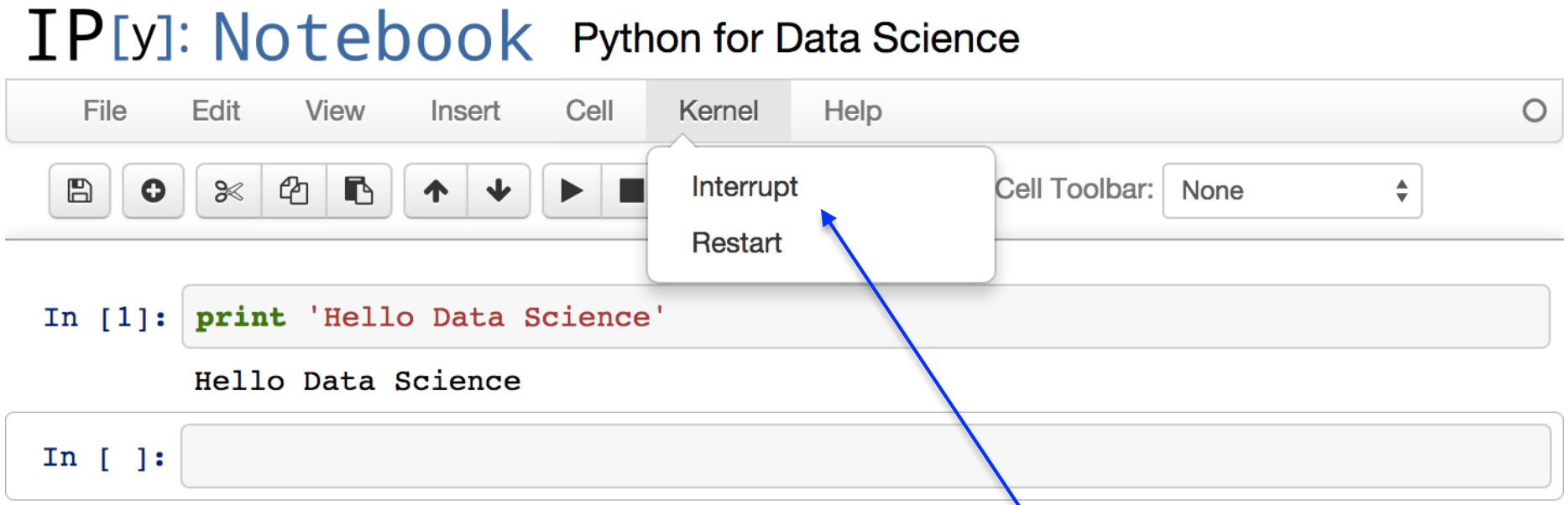


Re-order code blocks

iPython Notebook Basics

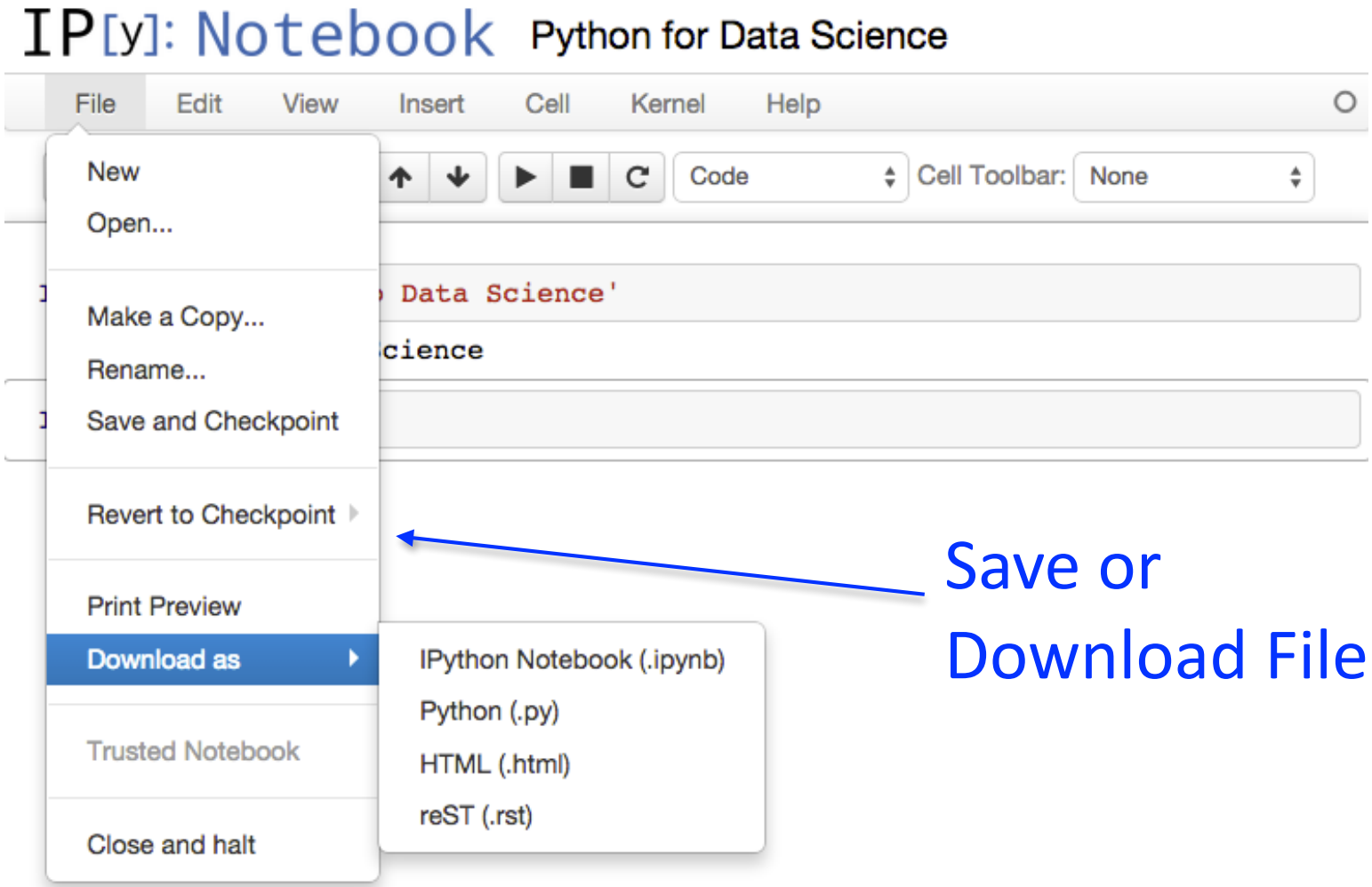


iPython Notebook Basics



Restart Notebook to Clear Memory

iPython Notebook Basics



Set Up for iPython Notebooks

- Create a folder on your Desktop Called "intro_python"
- Copy the files from my emails into the folder
- Open your terminal/command prompt and launch ipython notebook

Write Your First Python Code

Type in the first code block:

```
print “Hello Data Science”
```

Press Shift + Enter

Data Types

- **Numeric Types**
  - Integer (whole numbers)
  - Float (includes decimals)
  - Boolean (True/False)

- **Strings**
  - Text
  - Must be in single or double quotes

Python has function to return data type:
type(<value>)

Try Data Types

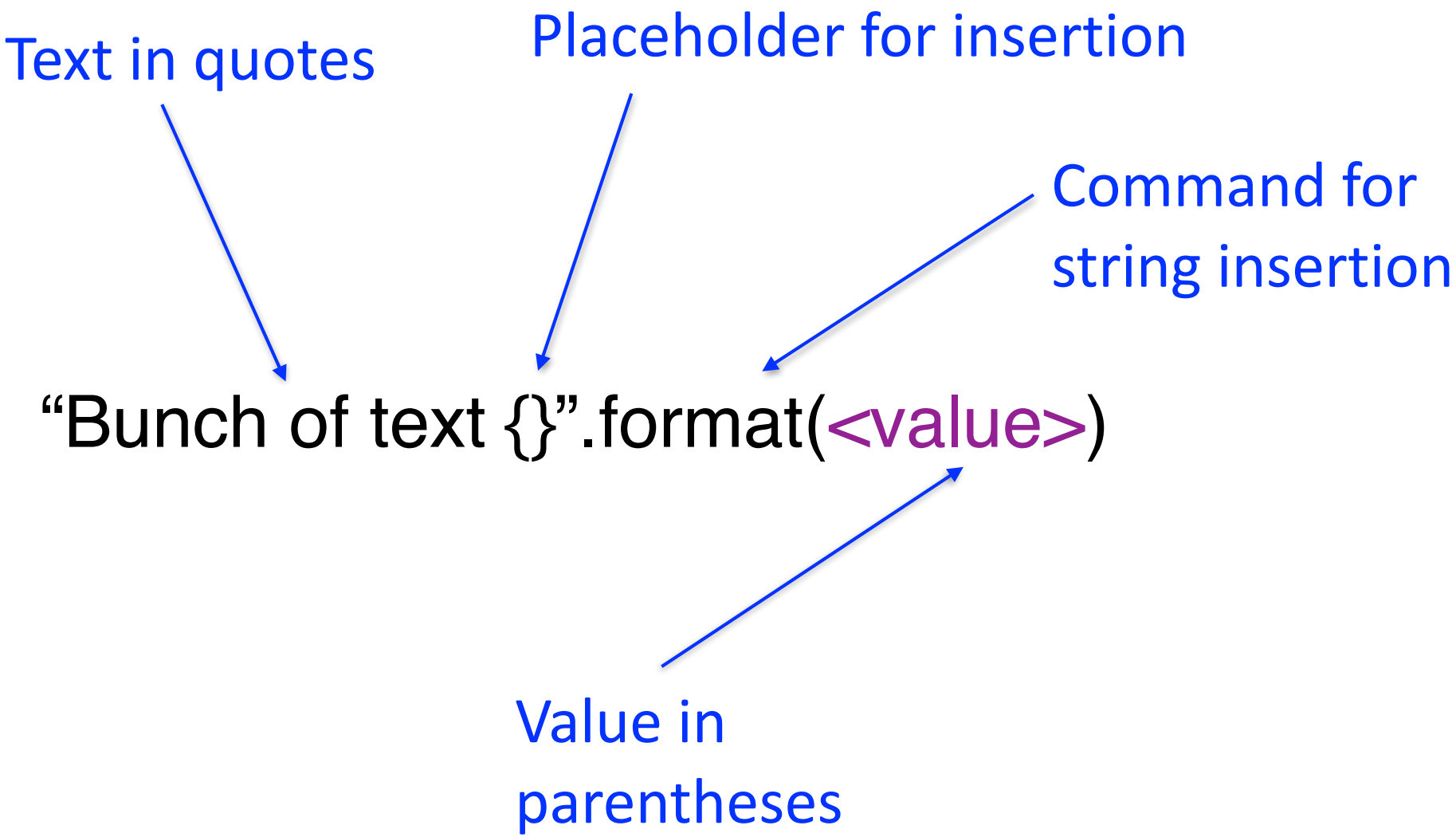
type(1)

type(2.5)

type(True)

type('string')

String Insertion Syntax



Try Basic String Insertion

```
"My name is {}".format('Craig')
```

```
name = "Waldo"
"Where in the world is {}".format(name)
```

Multiple Insertions

Multiple insertions require values in the brackets

```
place = “SF”
“{0} is in {1}”.format(name, place)
```

What happens when you change the order of the variables?

Using Names in Insertions

Names can be used in brackets

Easier to read but more verbose

```
“{nm} is in {pl}”.format(nm=name, pl=place)
```

```
"I<3{pl}.  I <3 {pl}!".format(pl=place)
```

Basic Math

Some operators are pretty obvious

```
5 + 5
```

```
3 * 7
```

Basic Math

Some are less intuitive

```
print “Hello ” + “World”
```

```
10 % 4  # modulo
```

```
10 ** 2  # exponent
```

```
1E3 + 1E-3  # exponent base 10
```

Variables

- Variables are objects that hold values
- Name variable using letters, numbers and underscore
- Special characters can’t be used for naming variables (e.g., [,*,@)
- Python commands can’t also be used as variable names
- Assign values to variables using single =
- You can re-assign values to variables

Assign Values to Variables

Create a few variables

```
x = 10
y = 5
z = 4
```

Try math with variables

```
x - y
```

```
x * y
```

Data Types in Math

Try dividing two integers

```
x / z
```

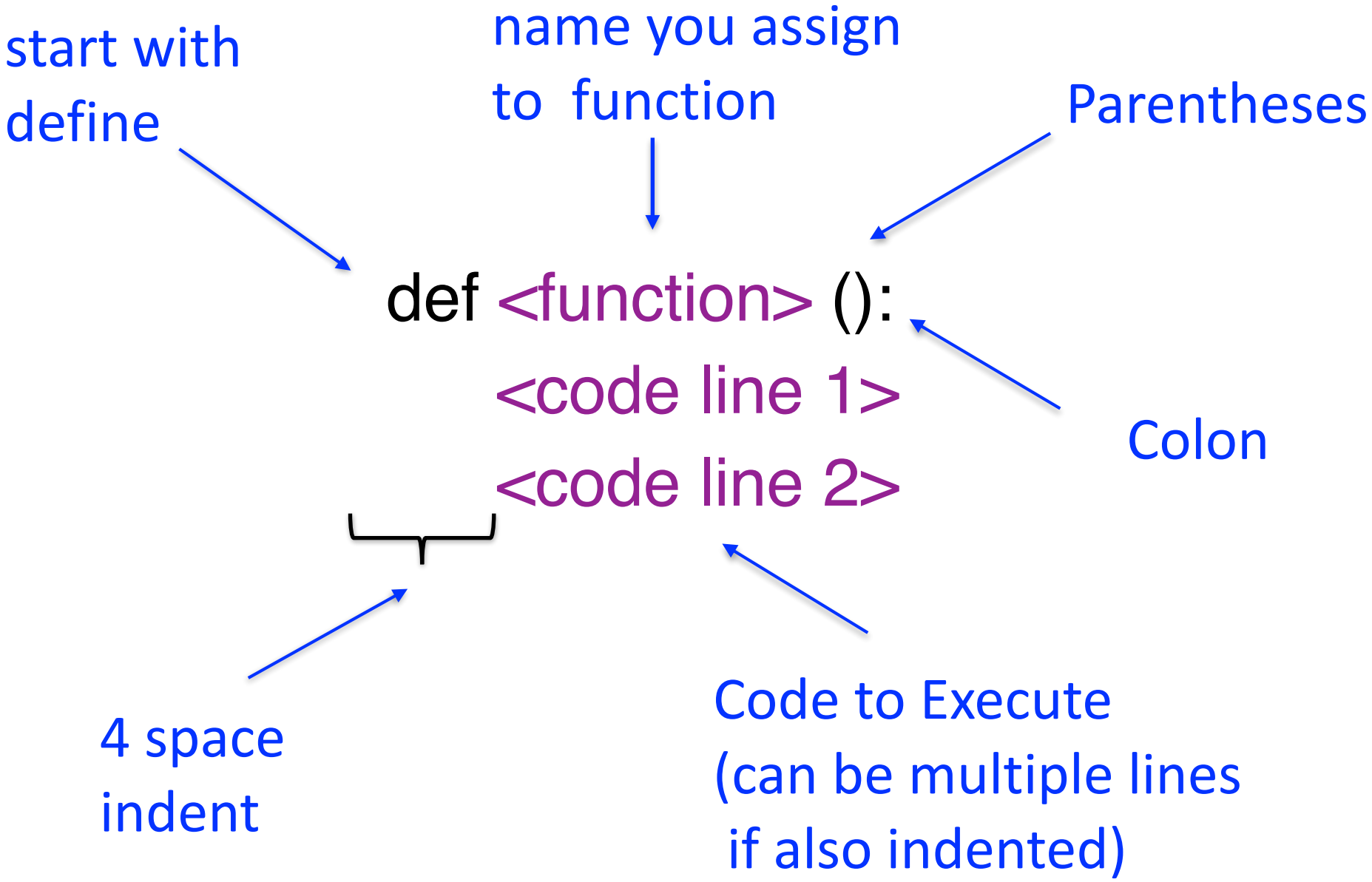
Now try using one float

```
x / 4.0
```

Functions

- Reusable snippets of code
- Define the function once
- Call the function to execute your code as many times as you like
- Can receive inputs and return results

Function Syntax



Create a simple function

Write a function

```
def simplest_function():
    print “I made a function”
```

Call the function

```
simplest_function()
```

Function with Input

Write a function that requires an input

```
def square(x):
    print x ** 2
```

Call the function

```
square(5)
```

Functions Can Return Values

Write a function that returns a result

```
def cube(x):
    return x ** 3
```

Call the function

```
y = cube(5)
print y
```

Line Continuation

- Sometimes code gets too long to write on one line
- Python automatically recognizes line continuation in specific cases like commas
- Backslashes (\ ) can be used to continue line of code

Line Continuation

Line continuation with commas

```
numbers = [1, 2, 3,
           4, 5, 6,
           7, 8, 9]
```

Backslashes can also be used for continuation

```
long_string = 'This is a really, really, really ' \
              + 'long sentence'
```

Instructions for Exercises

- Pair programming
  - Using only one computer
  - Take turns typing
  - Collaborate on solutions
- Save Examples for Future Reference
  - Add notes using # Comments
- Error Tracking
  - Create a text file to keep notes on your errors
- Trouble-shooting References
  - Online documentation
  - Stackoverflow / Google

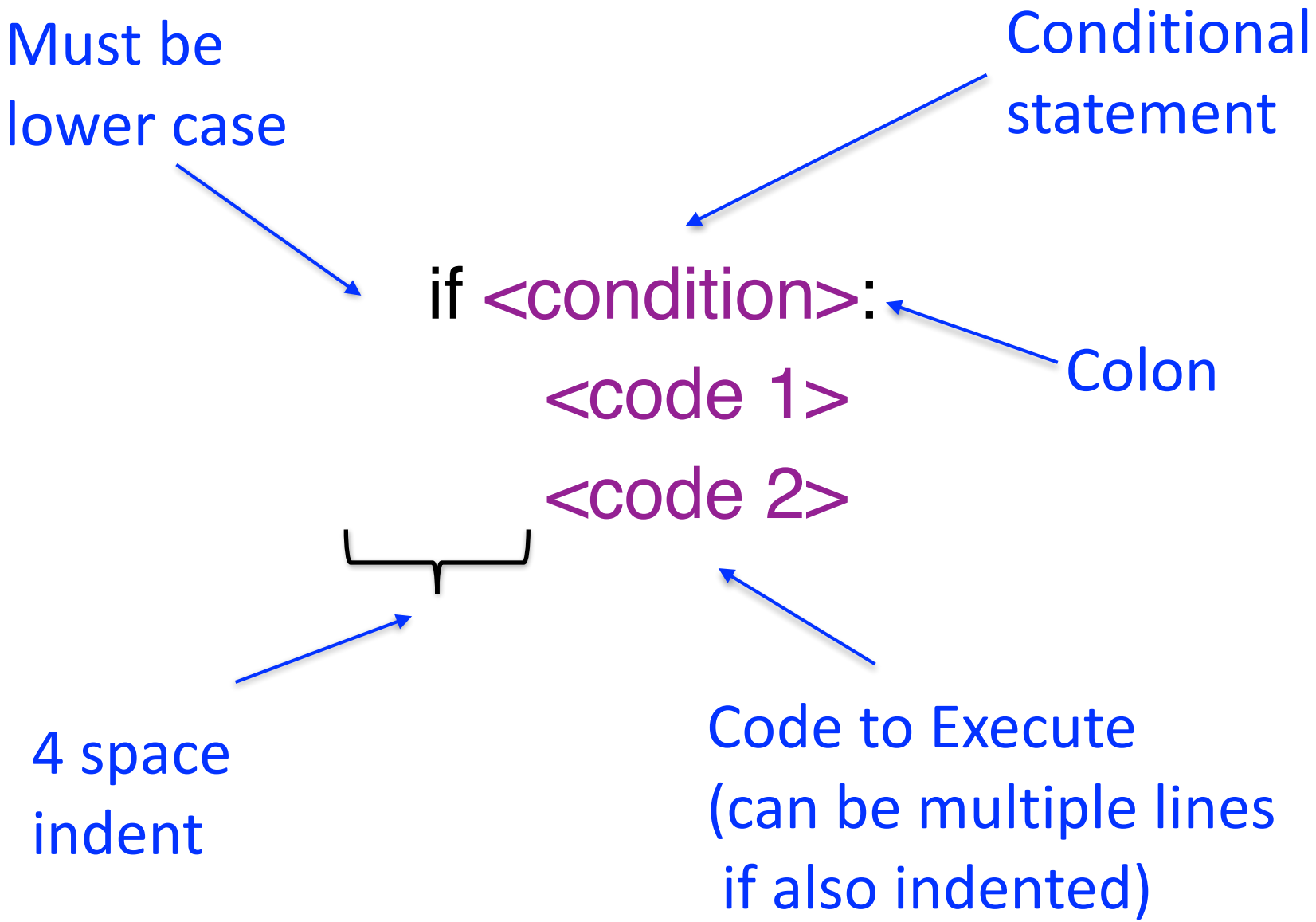
Exercise

1. Create a function that converts Celsius to Fahrenheit. Results should be accurate to at least one decimal point.
2. Update your function to return a sentence (string type) with the Celsius and Fahrenheit values inserted into the string.

If Statements

- Used to execute commands when defined conditions are met
- Contains a conditional statement that has a True/False value
- If statement is True then a series of commands will be executed
- If the statement is False then commands are skipped

If Statements Syntax



Conditional Statements

a == b	***Equal***
a != b	***Not Equal***
a > b	***Greater Than***
a >= b	***Greater Than or Equal***
a <= b	***Less Than or Equal***

Multiple Conditions

True and True = True

True & False = False

and, & are interchangeable

True or False = True

False | False = False

or, | are interchangeable

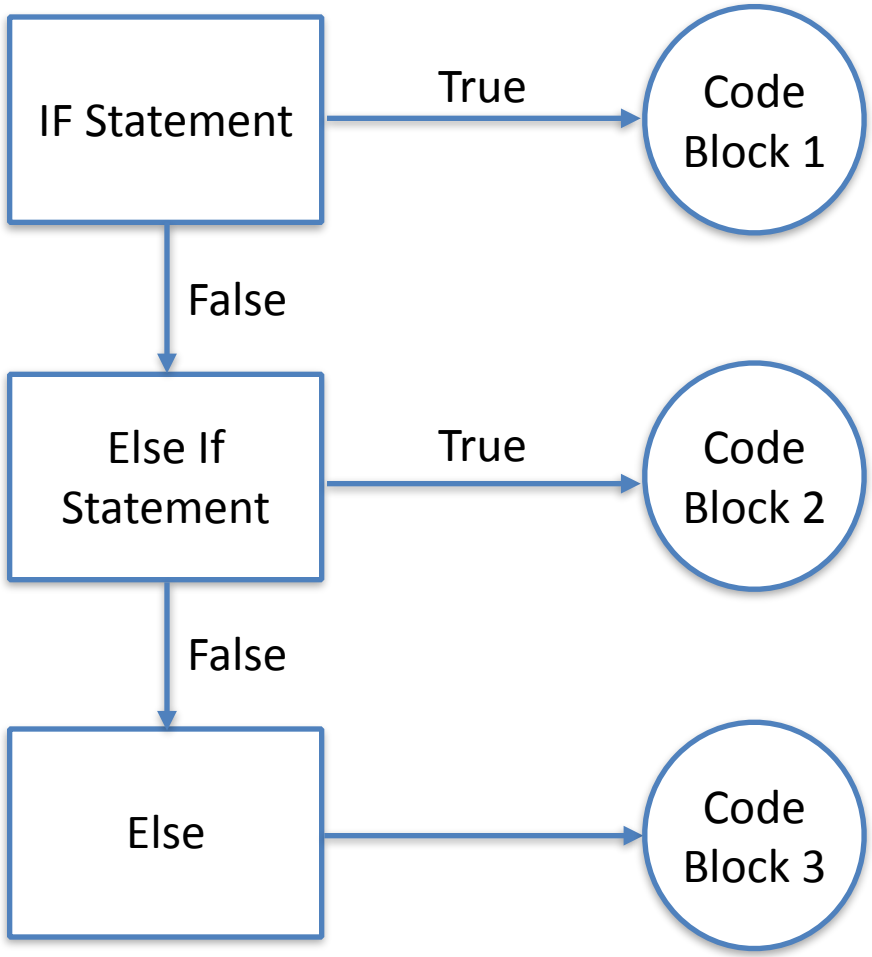
If Statement

Write a simple if statement

```
x = 3
if x > 0:
    print x
```

Else and Else If

Allow additional conditions and actions



Executes First True Statement

Else is catch all and must be at the end

Else If Statement

If statement with Else If

```
x = 3
if x > 0:
    print “{} is a positive number”.format(x)
elif x < 0:
    print “{} is a negative number”.format(x)
else:
    print “x equals 0”
```

Exercise

1. Create a function that checks the type of an input and returns a message stating whether it is numeric or not
2. Update your temperature function from the Python Fundamentals exercise to return an error message if a string is entered instead of a number

Lists, Tuples and Dictionaries

- Python has built-in objects that can hold multiple values
- Can be assigned to variables
- Has built-in methods
- Methods are functions for object

Lists

- Lists are ordered data containers
- Lists are defined with square brackets [ ]
- They can contain any type of objects
  - Mix of data types (e.g., integer, string, float)
  - Lists can even contain other lists
- List are mutable (you can edit them)
- Uses index to reference items in lists
- Lists can be empty

List Basics

Use brackets to define list

```
x = [1, 'b', True]
```

Use index position to reference items

```
print x[2]
```

Reassign values in a list

```
x[1] = 'a'
```

Indexing Lists

Create list of lists

```
a = [[1,2,3], 4, 5]
```

Use multiple indexes for lists within lists

```
print a[0][1]
```

Index from the end of the list

```
print a[-1]
```

Appending and Indexing

Append an item to a list

```
a.append('one more item')
```

Reference multiple items in a list

```
print a[2:4]
```

Open ended indexes go to the ends of lists

```
print a[:3]
```

Tuples

- Tuples are similar to lists
- Tuples are defined using parentheses ( )
- Only difference is that tuples are immutable (you can’t change them)
- Tuples with single value must have a comma (1,)

Tuple Basics

Use parentheses to define tuple

```
y = (1, ‘a’, 2.5)
```

Use index position to reference items

```
print y[0]
```

Try reassigning values in a tuple

```
y[0] = 2
```

Dictionaries

- Dictionaries are collections of key-value pairs
- Dictionaries are indicated by curly braces { }
- Values are looked up by key
- Dictionaries are unordered

Dictionary Basics

Create a dictionary

```
info = {'name': 'Bob', 'age': 54, 'kids': ['Henry', ‘Phil']}
```

Use key to reference a value

```
print info[‘name’]
```

Change the value for a key-pair

```
info[‘age’] = 55
```

Dictionary Methods

View all keys

```
info.keys()
```

View all values

```
info.values()
```

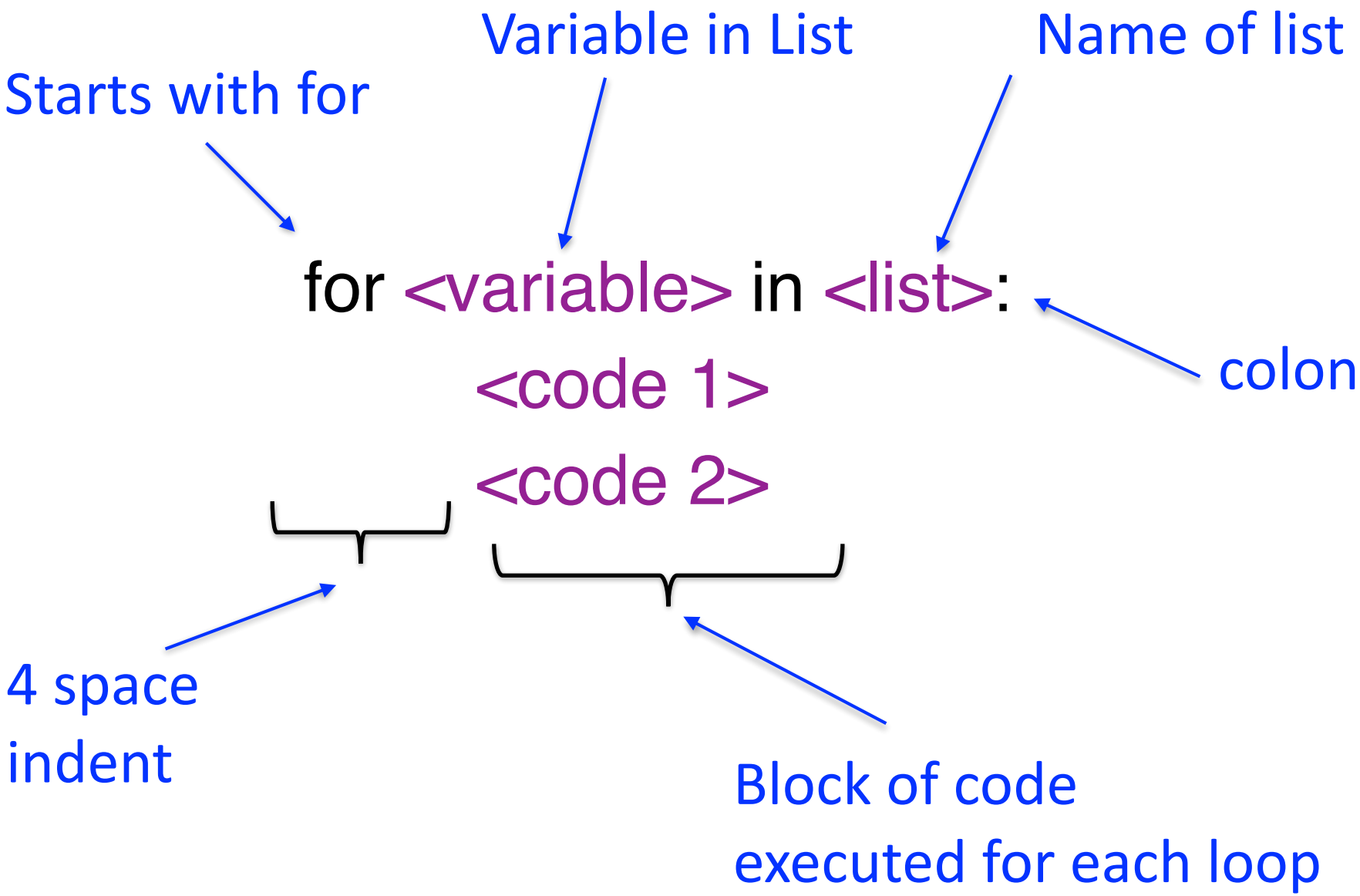
Check if a key exists

```
info.has_key('age')
```

For Loops

- Iterates through multiple values
- Commonly used to process values in a list
- Loop of code is executed for each item

For Loop Syntax



Functions Used with For

range(<integer>)

- Creates list of integers
- Starts with zero and each subsequent value is incremented by 1
- Returns list with length = input integer
- Last item in list is input -1 since list starts with zero

Basic For Loops

Create basic for loop

```
for x in [1,2,3]:
    print x
```

Create a for loop with range

```
for x in range(10):
    print x
```

For Loop with Multiple Values

Create for loop with multiple values

```
for x, y in [[1, 4], [2, 5], [3, 6]]:
    print x * y
```

For Loops with Empty List

Capture the all the results of a for loop

```
results = []
for x in [1,2,3]:
    squared = x **2
    results.append(squared)

print results
```

Exercise

1. Create a function that receives a list of numbers as an input, adds 1 to each number and returns the results as a list
2. Update your temperature conversion function from the Python Fundamentals exercise to accept a list of Celsius temperatures and return a list of Fahrenheit temperatures

Bonus:

Add error handling to your temperature conversion function.

Python Packages

- Data analytics packages are what make python so powerful
- Packages are just files of python code
- Importing packages allow you to use the functions from these files
- Most packages have online documentation and code examples

Common Packages for Data Science

Package	Usage
numpy	Scientific computing
pandas	Data slicing and manipulation
datetime	Manage date and time formats
matplotlib	Creating charts and graphs
scikit-learn	Machine learning
statsmodels	Statistics

Importing Packages

- Plain import statement:
  import <package name>

- Use a nickname:
  import <package name> as <nickname>

- Import a subset of the package:
  from <package> import <function>

- Avoid this technique, because it can create name-space conflicts
  from <package> import *

Import Packages

Let's import a package

```
import datetime
```

Use ipython magic to see function options. Type datetime. and press tab. Highlight time and press enter. Hit shift-tab

```
datetime.
```

Import Packages

Use a function from the datetime package

```
print datetime.time(1)
```

Import a package using a nickname

```
import datetime as dt
```

Call a function with the nickname

```
print dt.time(1)
```

Import Packages

Import a function from a package

```
from datetime import time
```

Directly call the time function

```
print time(1)
```

Datetime Package

Use the now function to get the current datetime stamp.

```
ts_now = dt.datetime.now()
print ts_now
```

Extract the day from the timestamp

```
print ts_now.day
```

Datetime Package

Extract some other datetime elements

```
print ts_now.year
print ts_now.month
print ts_now.minute
print ts_now.second
```

Create a timestamp for Christmas

```
ts_xmas = dt.datetime(2015,12,25)
print ts_xmas
```

Datetime Documentation

8.1.4. datetime Objects

A datetime object is a single object containing all the information from a date object and a time object. Like a date object, datetime assumes the current Gregorian calendar extended in both directions; like a time object, datetime assumes there are exactly 3600*24 seconds in every day.

Constructor:

class datetime.**datetime**(*year*, *month*, *day*[, *hour*[, *minute*[, *second*[, *microsecond*[, *tzinfo*]]]]])

The year, month and day arguments are required. *tzinfo* may be None, or an instance of a tzinfo subclass. The remaining arguments may be ints or longs, in the following ranges:

- MINYEAR <= year <= MAXYEAR
- 1 <= month <= 12
- 1 <= day <= number of days in the given month and year
- 0 <= hour < 24
- 0 <= minute < 60
- 0 <= second < 60
- 0 <= microsecond < 1000000

If an argument outside those ranges is given, ValueError is raised.

Other constructors, all class methods:

classmethod datetime.**today**()

Return the current local datetime, with tzinfo None. This is equivalent to datetime.fromtimestamp(time.time()). See also now(), fromtimestamp().

Timedelta Documentation

8.1.2. timedelta Objects

A `timedelta` object represents a duration, the difference between two dates or times.

class `datetime.`**`timedelta`**([*days*[, *seconds*[, *microseconds*[, *milliseconds*[, *minutes*[, *hours*[, *weeks*]]]]]]])

All arguments are optional and default to `0`. Arguments may be ints, longs, or floats, and may be positive or negative.

Only *days*, *seconds* and *microseconds* are stored internally. Arguments are converted to those units:

- A millisecond is converted to 1000 microseconds.
- A minute is converted to 60 seconds.
- An hour is converted to 3600 seconds.
- A week is converted to 7 days.

and days, seconds and microseconds are then normalized so that the representation is unique, with

- `0 <= microseconds < 1000000`
- `0 <= seconds < 3600*24` (the number of seconds in one day)
- `-999999999 <= days <= 999999999`

If any argument is a float and there are fractional microseconds, the fractional microseconds left over from all arguments are combined and their sum is rounded to the nearest microsecond. If no argument is a float, the conversion and normalization processes are exact (no information is lost).

If the normalized value of days lies outside the indicated range, `OverflowError` is raised.

Note that normalization of negative values may be surprising at first. For example,

```
>>> from datetime import timedelta
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

Timedeltas

Do math with the timestamps

```
ts_diff = ts_xmas - ts_now
print ts_diff
print type(ts_diff)
```

Strptime Function

Converts string to datetime

String to be converted

Starts with function

dt.datetime.strptime(<string> , <format>)

Format of string in quotes
(e.g., “%Y-%m-%d”)

Format Examples:

%Y	Year
%m	Month
%d	Day

Strptime Function

Convert a string to datetime format

```
date_str = "01-10-2015"
date_ts = dt.datetime.strptime(date_str,
      "%m-%d-%Y")
print date_ts
print type(date_ts)
```

Strptime Documentation

Directive	Meaning	Example	Notes
%a	Weekday as locale's abbreviated name.	Sun, Mon, ..., Sat (en_US); So, Mo, ..., Sa (de_DE)	(1)
%A	Weekday as locale's full name.	Sunday, Monday, ..., Saturday (en_US); Sonntag, Montag, ..., Samstag (de_DE)	(1)
%w	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	0, 1, ..., 6	
%d	Day of the month as a zero-padded decimal number.	01, 02, ..., 31	
%b	Month as locale's abbreviated name.	Jan, Feb, ..., Dec (en_US); Jan, Feb, ..., Dez (de_DE)	(1)
%B	Month as locale's full name.	January, February, ..., December (en_US); Januar, Februar, ..., Dezember (de_DE)	(1)
%m	Month as a zero-padded decimal number.	01, 02, ..., 12	
%y	Year without century as a zero-padded decimal number.	00, 01, ..., 99	
%Y	Year with century as a decimal number.	1970, 1988, 2001, 2013	
%H	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, ..., 23	
%I	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, ..., 12	
%p	Locale's equivalent of either AM or PM.	AM, PM (en_US); am, pm (de_DE)	(1), (2)
%M	Minute as a zero-padded decimal number.	00, 01, ..., 59	
%S	Second as a zero-padded decimal number.	00, 01, ..., 59	(3)
%f	Microsecond as a decimal number, zero-padded on the left.	000000, 000001, ..., 999999	(4)
%z	UTC offset in the form +HHMM or -HHMM (empty string if the the object is naive).	(empty), +0000, -0400, +1030	(5)
%Z	Time zone name (empty string if the object is naive).	(empty), UTC, EST, CST	
%j	Day of the year as a zero-padded decimal number.	001, 002, ..., 366	
%U	Week number of the year (Sunday as the first day of the week) as a zero padded decimal number. All days in a new year preceding the first Sunday are considered to be in week 0.	00, 01, ..., 53	(6)

Strftime Function

Convert a datetime object to a string

```
new_date_str = dt.datetime.strftime(
        date_ts,"%H:%M:%S %m-%d-%Y")
print new_date_str
```

Exercise

1. Create a variable called future with a dateime value of January 15, 2016 at 5:30pm
2. Create a variable with a timedelta equal to the difference between future and now
3. Print the number of seconds from the timedelta

Bonus:

Try converting strings to datetime objects and datetime objects back to strings. What are some interesting datetime formats you can find in the documentation?

Coding Best Practices

PEP-8 Style Guide

- https://www.python.org/dev/peps/pep-0008/
- maximum line length of 79 characters
- indentation
- line continuation
- commenting

Hard to Use at First But Review Once a Month and You'll Incrementally Improve

Functional Programming

- Popular programming technique for data science
- Build a set of tools that you can reuse - great way to practice
- Functional programs should only take inputs and provide outputs
- Complex programs can be broken down as a collection of smaller functions
- Easier to test than large complex programs

Coding Best Practices

Documentation

- Write a simple description of your functions in first line after defining function
- Include descriptions of each input parameter and return value (e.g., name, description, data type)
- Use intuitive variable names

Debugging Tips

- Test early and often
- print command is your best friend
- Identify common mistakes and use as checklist for debugging
- Use google and stackoverflow
- Check documentation
- Ask for help

Next Steps

- Practice, Practice, Practice
  - Find a fun project
  - Get a pair programming buddy
  - Don’t Be Afraid to Ask for Help
- Join a Python Meetup
  - San Francisco Python
  - Bay Area Python Interest Group (BayPIGgies)
- Keep Learning
  - Pycon videos on YouTube
  - Python for Data Science Bootcamp
- Keep in Touch