**Jupyter slideshow:** This notebook can be displayed as slides. To view it as a slideshow in your browser, type following in the console:

> jupyter nbconvert [this\_notebook.ipynb] --to slides --post serve

To toggle off the slideshow cell formatting, click the CellToolbar button, then View > Cell Toolbar > None.

For more help, check out [this tutorial](https://drive.google.com/open?id=17q01buf7YFuB4yF8cFjmnc_ZARQWOljGhlHs99i9X0c).

In [2]:

print('hello world')

hello world



Introduction to Python Fundamentals[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Introduction-to-Python-Fundamentals)

*Authors: Kiefer Katovich (San Francisco), Dave Yerrington (San Francisco), Joseph Nelson (Washington, D.C.), Sam Stack (Washington, D.C.)*

Learning Objectives[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Learning-Objectives)

*After this lesson, you will be able to:*

* Define what a type is and what kinds exist in Python.
* Define a function and identify common functions in Python.
* Define control flow and some common examples in Python.

Lesson Guide[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Lesson-Guide)

* [Survey (5–10 min)](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#survey1)
* [Common Python Types (10 min)](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#types-def)
* [Common Types Code-Along (20 min)](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#types-codealong)
* [Common Python Functions and Control Flow (10 min)](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#functions-def)
* [Common Python Functions Code-Along (20 min)](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#functions-codealong)
* [Recap and Requests (5–10 min)](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#recap-requests)

**Instructor Note:** We also include a couple of sections for quizzes (one after types and one after functions) — these can be used as checks for understanding either formally — by asking students to complete them and checking their work — or informally, by having students explain their reasoning to the class.

Survey (Pre-Work Comprehension)[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Survey-(Pre-Work-Comprehension))

Please break into groups of three or four and discuss the following over the the next few minutes:

* What are 3–4 takeaways from the pre-work that were new to you?
* What are 3–4 (as of yet) unanswered questions you had from the pre-work?

**Instructor Note:** This check is intended to identify students' takeaways from the pre-work and understand what questions they have coming into the course. You may note their answers or write them on the whiteboard for further facilitated discussion.

Common Python Types[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Common-Python-Types)

**In Python, what do you understand a "type" to be?**

**Instructor Note:** Guide students to think about different ways that data can be encoded and what happens when you assign data to a variable.

**Instructor Note:** Depending on how in depth you would like to get on the subject of types, you could make use of this material here:

There are several *standard* data types within Python (and typically any programming language, although it varies as to what is or isn't in a type for each).

A type tells the computer what can or can't be done with a given piece of information in Python (or whichever programming language you are using). For example, we can **add** and **subtract** numbers and we can **count** the number of letters in a word. Rather than try (and fail!) to add two words together or count the number of numbers in a number, types let the computer know efficiently what can and cannot be done in a given situation.

Different languages have different requirements for typing, known broadly as *static versus dynamic* typing and *strong versus weak typing*:

* **Static typing**: When we create a variable, we tell the program what type it is, and it stays that way for the evaluation of the program.
* **Dynamic typing**: When we create a variable, it can be of any type and can be changed if we redefine it in the program.
* **Strong typing**: A variable's type determines what can and cannot be done with it (such as dividing a word).
* **Weak typing**: You can try to do anything with that variable and, if it can be converted in the process to make that operation work, the language will do so.

Python is a *dynamic, strongly typed* language — while we can redefine a variable's type as we work with it, Python won't change types on us unless we explicitly tell it to do so. This is a design choice of Python; different languages do things differently (if you pick up R, C++, or Java in the future, expect differences).

**In your own words**, discuss what *static*, *dynamic*, *strong*, and *weak* typing is with a partner.

Types in Python[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Types-in-Python)

**What are some of the basic types in Python? Are there Python types that contain multiple elements?**

**Instructor Note:** Guide students toward creating a taxonomy of types similar to the following.

There are two basic groups of types in Python — single elements (one item or piece of information) and collections (groups of things, which can be either single elements themselves, other collections, or a mix of both).

**Single Elements**

* **Integers:** Whole numbers ranging from negative infinity to infinity, such as 1, 0, -5, etc.
* **Floats:** Short for "floating point number;" usually used with decimals, such as 2.8 or 3.14159.
* **Strings:** A set of letters, numbers, or other characters, e.g., "The fox is quick." — any set of values that contains a non-numeric character.

**Collections**

* **Tuples:** An ordered sequence with a fixed number of elements; e.g., in x = (1, 2, 3), the parentheses makes it a tuple. x = ("Kirk", "Picard", "Spock") — once you've defined this, you can't change it.
* **Lists:** An ordered sequence without a fixed number of elements, e.g., x = [1, 2, 3]. Note the square brackets. x = ["Lord", "of", "the", "Rings"] — this can be changed as you like.
* **Dictionaries**: An unordered collection of key-value pairs, e.g., x = {'Mark': 'Twain', 'Apples': 5}. To retrieve each value (the part after each colon), use its key (the part before each colon). For example, x['Apples'] retrieves the value 5.

**In your own words, what is the difference between a list and a dictionary? Can you think of any real-life examples of lists or dictionaries?**

Throughout this lesson, we'll review each data type more in depth and discuss common ways of interacting with each of them.

* [Python's basic data types](https://en.wikiversity.org/wiki/Python/Basic_data_types).

**Instructor Note:** Guide students to think of real-world items like the ones below that are similar to a list or dictionary. This is also a good place to discuss when these collections are used most frequently.

Typically, lists and dictionaries are most popular because of their ease of use and applicability to numerous situations (by contrast, because tuples are immutable/cannot be modified; they are used more sparingly). Lists have an inherent order (the first element is at index 0, the second element is at index 1, etc.) and we can call each element by that ordinal number (such as x[0] or x[100]). Dictionaries do not have an order (so x[0] will fail), but they use the name of the key to return that element.

**Example**: Think about the difference between a sign-up list and an address book:

* **Sign-up list**: We would refer to the order of who signed up when (i.e., "the eighth person to sign up").
* **Address book**: We would refer to a person by looking for their name (i.e., "the contact info for Bill Personson").

Variables[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Variables)

Variables are names that have been assigned to specific values or data. These names can be almost anything you want, but there are some restrictions and best practices.

**In your own words, can you think of some variable names that might be restricted within Python?**

**Instructor Note:** List their rules on the whiteboard and then bring up some of the concepts below.

Restrictions[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Restrictions)

* Variable names cannot be just a number (i.e., 2, 0.01, 10000).
* Variables cannot be assigned the same name as a default or imported function (i.e., 'type', 'print', 'for').
* Variable names cannot contain spaces.

Best Practices[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Best-Practices)

* Variable names should be lowercase.
* A variable's name should be representative of the value(s) it has been assigned. For example, instead of a or money, something like money\_usd tells the user both what is stored in that variable *and* what units it's in.

Common Types Code-Along[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Common-Types-Code-Along)

In this section, we'll practice establishing some types and common practices. To run each cell, use shift+enter or the run button in the Jupyter Notebook toolbar.

Some common reasons why we need to keep types in mind:

1) Different types can lead to different results:

- `1 + 1` results in `2`, while `'1' + '1'` results in the string `11`.

2) Operations may not work with specific types:

- `len('a word')` will return the number of characters in a word, while `len(25)` will return an error because numbers do not have a length.

In [1]:

# Assigning a float:

x = 1.0

type(x)

Out[1]:

float

In [ ]:

# Assigning an int:

y = 1

type(y)

In [ ]:

# Assigning a string:

z = '1'

type(z)

**Remember that, when we're assigning variables, we are not stating that "x equals 1," we are stating that "x has been assigned the value of 1."**

Operators[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Operators)

Operators can be used in a mathematical sense to calculate (or create) the sum, difference, product, or quotient of values or variables.

Note that print() below will print out the values of whatever is inside of the parentheses.

In [ ]:

# Addition:

print(1 + 2)

# Subtraction:

print(1 - 2)

# Multiplication:

print(1 \* 2)

# Division:

print(1 / 2)

There is also // division, whose output will be the rounded-down whole number.

In [ ]:

# Division of float numbers:

print(3.0 // 2)

print(-3.0 // 2)

In [ ]:

# Exponent power operator:

2 \*\* 2

In [ ]:

# The modulo operator can be used to get the remainder — what's left over after the term has been cleanly divided:

5%2.

Booleans and Boolean Evaluation Operators[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Booleans-and-Boolean-Evaluation-Operators)

**In your own words, define a boolean.**

**Instructor Note:** Guide students to the following definition:

Booleans exist as either true or false and are generally used as a means of evaluation.

Using Booleans[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Using-Booleans)

Booleans are frequently used to filter data or conditions. Sometimes, we may want all countries with populations greater than 4,000,000 or all people named Bob. Both of these result in a True or False condition that split our data into the groups we want.

In Python, there are several built-in commands for deciding how to filter results:

* and: Are both A and B true?
* not: Is A the same as B?
* or: Is A or B true?

In [ ]:

True and False

In [ ]:

not False

In [ ]:

True or False

**Comparisons**

* Less than: <
* Greater than: >
* Less than or equal to: <=
* Greater than or equal to: <=
* Equals: ==
* Does not equal: !=

In [ ]:

2 > 1, 2 < 1, 2 > 2, 2 < 2, 2 >= 2, 2 <= 2

In [ ]:

# Equality:

[1,2] == [1,2], [1,2] != [2,1]

In [ ]:

[1,2] == [1,2] and [2,2] == [2,2]

Now You Try![¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Now-You-Try!)

With a partner, create three comparisons using !=, >=, and <.

Strings[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Strings)

**What are strings? How would we use them? Can you think of any examples of strings?**

**Instructor Note:** Guide your students to the following definition:

Strings are essentially any character combination in between quotes. They are most often used as a way of storing text. Strings are used frequently, because most of the data that humans create are text-based, such as restaurant reviews or emails.

In [ ]:

s = "Hello world"

type(s)

Strings have a lot of associated methods and attributes that allow us to better understand and manipulate them.

**In your own words, why would we want to manipulate or change strings?**

**Instructor Note:** Some examples include:

* Fixing misspelled words.
* Changing casing (upper, lower).
* Looking for specific words.

In [ ]:

# Finding the length of the string:

len(s)

In [ ]:

# Replacing an element of a string:

s2 = s.replace("world", "test")

print(s2)

String Indexing[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#String-Indexing)

In some cases, we may want a part of the string (like the first character for alphabetizing or categorizing). Indexing helps us do that.

We can extract characters at specific index locations in a string using indexing.

In [ ]:

# Indexing the first (index 0) character in the string:

s[0]

The number you enter after the variable name in brackets (the [0]) is called the index (its plural is indices).

*Counting in Python and many other programming languages begins at zero, as opposed to one. This is called zero-based indexing.*

In [ ]:

# This is called "splicing." We start at the left index

# and go up to but not include the right index.

# Objects at indexes 0, 1, and 2:

s[0:3]

Most ranges, or functions with ranges, have upper ends that are not inclusive. So, a range of [0:5] starts at 0 and stops before 5.

A good mental trick is to look at something like [5:25] and say out loud "Starting at five and going up to (but not including) 25."

In [ ]:

# From index 6 up to the end of the string:

s[6:]

In [ ]:

# No start or end specified:

s[:]

In [ ]:

# Can we index from the right side?

s[-1]

In addition to specifying a range, you can include a step size or character skip rate. This might be helpful if you want every other letter, for example.

These indexing methods can also be used on lists, where asking for every other number might be a good use case.

In [ ]:

# Every second character starting at 0 and ending at 10:

s[0:10:2]

In [ ]:

# Define a step size of 2; i.e., every other character:

s[::2]

In [ ]:

# The same, but for a list of numbers:

[0, 1, 2, 3, 4, 5, 6][::2]

Concatenating[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Concatenating)

**In your own words, what is concatenating? When might you use it?**

To add two strings together, type the first string, a + sign, and then the second string.

**Instructor Note:** One easy example is adding a greeting to someone's name: 'Hello ' + name.

In [ ]:

x = 'Hello'

y = 'world'

x + y

In [ ]:

# Conversion from int to str is required!

dice\_roll = 3

print('You rolled a ' + str(dice\_roll) + '.')

Now You Try![¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Now-You-Try!)

Create your own string of at least 12 characters or more and: 1) Test to make sure that it is at least 12 characters long.   
2) Replace all of the vowels with the string vowel.   
3) Use concatenation to add another string to it.

Lists[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Lists)

**What are some examples of lists? What do you remember from before?**

Lists can be composed of ints, floats, strings, or other lists, as well as other data types we haven't covered yet.

In [ ]:

l = [1, 2, 3, 4]

print(type(l))

print(l)

In [ ]:

# The a variable's contents can be reassigned to another variable:

a = l

In [ ]:

print(a)

In [ ]:

# List of strings:

names = ['Carol', 'Anne', 'Jessica']

print(names)

Methods[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Methods)

Many types have what are known as "methods:" built-in functionality that allows them to do certain things. We've already seen a couple, such as the .replace() method, which lets you replace words in strings.

Lists also have several methods that allow us to alter them, such as the .append() method, which allows us to add another element to the end of a list.

In [ ]:

names.append('Michelle')

names

Lists can indexed the same way strings — this allows us to target a specific value or range of values in a list without having to create a new one.

In [ ]:

print(names[1:3])

print(names[::2]) # Increments the index by 2 each time (skips alternate elements).

In [ ]:

# We can slice a value in a list as well:

names[1][1:]

Note that we always read indexing from left to right. In the example above, the interpreter looks up names and gets the first element, which is the string "Anne". Then, the slice ([1:]) adds the first index of that string to the end of the original string, evaluating to "nne".

Interestingly, the following works in the same way. Instead of having to look up the value of names, the list is directly specified (just read the line from left to right!).

In [ ]:

['Carol', 'Anne', 'Jessica', 'Michelle'][1][1:]

In [ ]:

# Lists don't have to be the same type:

l = [1, 'a', 1.0, 1-1j]

print(l)

In [ ]:

# We can create a list of values in a range using the range() function:

start = 10

stop = 30

step = 2

print(type(range(start, stop, step)))

# range() produces a "generator," which is beyond the scope of this introduction!

# It is often convenient to have the generator

# generate all of its values by converting it to a list:

list(range(start, stop, step))

Use the .insert() method to add values at specific indices.

In [ ]:

names.insert(2, 'Ellen')

names

The .remove() method can be used to remove specific values if they appear in a list.

In [ ]:

names.append('Jeremy')

print(names)

names.remove('Jeremy')

print(names)

Now You Try![¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Now-You-Try!)

Create a list of five elements and do the following: 1) Print the last three elements.   
2) Insert two new elements at index 2 and append one element to the end.   
3) Remove one element of your choice.   
4) Print every other element in your list.

Tuples[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Tuples)

Tuples are similar to lists in that they store a sequence of various separate values. However, tuples are not mutable in that, once they are created, their values cannot be changed.

**In your own words, why would creating something that cannot be changed later be helpful?**

**Instructor Note:** Guide students to think about safety — what if you absolutely needed to make sure that one or two numbers or values stayed the same?

In [ ]:

point = (10, 20)

print(point)

print(type(point))

In [ ]:

# They can be sliced, just like lists and strings:

point[0]

Dictionaries[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Dictionaries)

Dictionaries are a non-ordered Python data type. Instead of using an ordered index to access data stored in a dictionary, we use a system of key-value pairs.

**In your own words, why would we use this when we could just use a list?**

* A key is similar to a variable name.
* A value is similar to the value assigned to the variable.
* Curly braces ({ }) enclose dictionaries. The first input in a dictionary pair is the "key." The second input in a dictionary pair is the "value." Remember to make key:value pairs!

The general format looks like this:

**Instructor Note:** Some answers might include:

* When there are lots of elements and we don't want to remember which element number is which.
* When we don't care about the order in which things happen.
  + As opposed to a list, which is kind of like a queue.

In [ ]:

params = {'key1' : 1.0,

'key2' : 2.0,

'key3' : 3.0,}

print(type(params))

print(params)

The keys stay the same, but the values are changeable. You can also only have one occurrence of a key in a dictionary, but you can have all of the values be the same.

In [ ]:

# Value for parameter2 in the params dictionary:

params['key2']

In [ ]:

# Adding a new dictionary entry:

params['key4'] = 'D'

In [ ]:

print(params)

In [ ]:

# Reassigning the value of a key-value pair in the dictionary:

params['key1'] = 'A'

params['key2'] = 'B'

In [ ]:

print('Key 1 = ' + str(params['key1']))

print('Key 2 = ' + str(params['key2']))

print('Key 3 = ' + str(params['key3']))

print('Key 4 = ' + str(params['key4']))

In [ ]:

# Dictionaries also have methods.

# Convert a dictionary to a list of tuples (key-value pairs).

# This is later used to conveniently loop through a dictionary:

list(params.items())

Quiz: Types[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Quiz:-Types)

*Identify the variable types of the following five items:*

* 1
* -1.0
* 1,000,000
* '10'
* ('twenty-four')

*Create a list of all numbers between 1 and 100, inclusive, using the range() function discussed above. Can you slice the list so that we see every fifth number, starting at 4 and ending at 82?*

**Instructor Note**: Feel free to make this quiz as informal or formal as you need for your class.

Common Python Functions and Control Flow[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Common-Python-Functions-and-Control-Flow)

**Instructor Note:** This is a basic introduction to control flows and functions. Depending on how well prepared your students are, you can consider introducing while, keyword arguments, or some more advanced function generation.

In this section, we're going to tackle some common design patterns in Python. The first is the concept of control flow — this is how our programs will return different results based on specific input. Second, we'll cover basic functions — these let us create snippets of code that we can call later in a script, which creates code that's easier to read and maintain. Remember, we're going to be reading code much more often than writing it!

if… else Statements[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#if…-else-Statements)

In Python, indentation matters! This is especially true when we look at the control structures in this lesson. In each case, a block of indented code is only run some of the time. There will always be a condition in the line preceding the indented block that determines whether the indented code is run or skipped.

if Statement[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#if-Statement)

The simplest example of a control structure is the if statement. We start with if, followed by something that can evaluate to True or False (such as any of the comparison operators we discussed earlier).

In [ ]:

if 1 == 1:

print('The integer 1 is equal to the integer 1.')

print('Is the next indented line run, too?')

In [ ]:

if 'one' == 'two':

print("The string 'one' is equal to the string 'two'.")

print('---')

print('These two lines are not indented, so they are always run next.')

Notice that, in Python, the line before every indented block must end with a colon (:). In fact, it turns out that the if statement has a very specific syntax:

if <expression>:

<one or more indented lines>

When the if statement is run, the expression is evaluated to True or False by applying the built-in bool() function. If the expression evaluates to True, the code block is run; otherwise, it is skipped.

Now You Try![¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Now-You-Try!)

Create your own string called test\_string, then fill in the blanks here to create an if... else statement for whether or not the first character in test\_string is a lowercase a.

In [ ]:

test\_string = '' # Fill in with your choice of string.

if test\_string: # Change this section to check if the first character in test\_string is a lowercase a.

print('Begins with a')

else:

print('Does not begin with a')

if ... else[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#if-...-else)

In many cases, you may want to run some code if the expression evaluates to True and some other code if it evaluates to False. This is done using else. Note how it is at the same indentation level as the if statement, followed by a colon, followed by a code block. Let's see it in action.

In [ ]:

if 50 < 30:

print("50 < 30.")

else:

print("50 >= 30.")

print("The else code block was run instead of the first block.")

print('---')

print('These two lines are not indented, so they are always run next.')

if ... elif ... else[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#if-...-elif-...-else)

Sometimes, you might want to run one specific code block out of several. For example, perhaps we provide the user with three choices and want something different to happen with each one.

elif stands for else if. It belongs on a line between the initial if statement and an (optional) else.

In [ ]:

health = 55

if health > 70:

print('You are in great health!')

elif health > 40:

print('Your health is average.')

print('Exercise and eat healthily!')

else:

print('Your health is low.')

print('Please see a doctor now.')

print('---')

print('These two lines are not indented, so they are always run next.')

This code works by evaluating each condition in order. If a condition evaluates to True, the rest are skipped.

**Let's walk through the code.** First, we let health = 55. We move to the next line at the same indentation level — the if. We evaluate health > 70 to be False, so its code block is skipped. Next, the interpreter moves to the next line at the same outer indentation level, which happens to be the elif. It evaluates its expression, health > 40, to be True, so its code block is run. Now, because a code block was run, the rest of the if statement is skipped.

for Loops[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#for-Loops)

One of the primary purposes of using a programming language is to automate repetitive tasks. One example is the for loop.

The for loop allows you to perform a task repeatedly on every element within an object, such as every name in a list.

Let's see how the pseudocode works:

*# For each individual object in the list*

*# perform task\_A on said object.*

*# Once task\_A has been completed, move to next object in the list.*

Let's say we wanted to print each of the names in the list, as well as "is Awesome!" In this case, we'd create a temporary variable for each element in the collection (for name in names would put each name, in sequence, under the temporary variable name) and then do something with it.

In [ ]:

names = ['Rebecca Bunch', 'Paula Proctor', 'Heather Davis']

for name in names:

print(name + ' Is Awesome!')

We can also combine if... else statements and for loops:

In [ ]:

for name in names:

if name == 'Paula Proctor':

print(name + ' Is REALLY AWESOME!')

else:

print(name + ' Is Awesome!')

Now You Try![¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Now-You-Try!)

Create a new if... elif... else and for loop combination, using a list of your own choice.

Functions[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Functions)

**When would you want to call the same code over and over again? What benefit does that have in programming?**

**Instructor Note:** Guide students to think about how using functions can make programming easier and ensure that we will only need to make changes/define options once in a script instead of multiple times.

Similar to the way we can use for loops as a means of performing repetitive tasks on a series of objects, we can also create functions to perform repetitive tasks. Within a function, we can write a large block of action and then call the function whenever we want to use it.

Let's write some pseudocode, which is code that Python will not run successfully, but illustrates the basic idea without worrying about correct syntax:

*# Define the function name and the requirements it needs.*

*# Perform actions.*

*# Optional: Return output.*

A function is defined like this:

**def** function\_name(arguments):

*# Do things here.*

**return** value

We start with def and the name of our function, then a set of parentheses. The terms we put in the parentheses will be passed into the function and stored in those variables. Finally, if we want to store the results of the function, we use return, which will let us take some value and store it once the function has run, like this:

x = function\_name(20)

Whatever follows return when the function is defined will be passed out of the function and stored in x.

Let's create a function that takes two numbers as arguments and returns their sum, difference, and product.

In [ ]:

def arithmetic(num1, num2):

print(num1 + num2)

print(num1 - num2)

print(num1 \* num2)

arithmetic(3,5)

Common Functions Code-Along[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Common-Functions-Code-Along)

In this section, we'll run through some basic functions and how we might use them.

**Instructor Note:** In this section, we'll run through some basic function generation so students can see the process of how to create and use functions. It can be helpful to walk through the code and ask students what each line does. A good suggestion would be to delete the latter sections of code and create them for your students in person so that they can see your thought process.

Write a function that takes the length of a side of a square as an argument and returns its area.

In [ ]:

def area\_square(length):

return length\*\*2

print(area\_square(4))

Write a function that takes the height and width of a triangle and returns its area.

In [ ]:

def area\_triangle(height, width):

return height + (0.5\*width)

print(area\_triangle(2, 6))

Write a function that takes a string as an argument and returns a tuple consisting of two elements:

* A list of all of the characters in the string.
* A count of the number of characters in the string.

In [ ]:

def list\_and\_count(word):

list\_of\_characters = []

for char in word:

list\_of\_characters.append(char)

return list\_of\_characters, len(word)

print(list\_and\_count('Lisa Simpson'))

Write a function that takes two integers, passed as strings, and returns the sum, difference, and product as a tuple (with all values as integers).

**Instructor Note:** If students are well-versed, you can include try and except here. This is also a good place to challenge your students to explain what would happen if they passed in something that could not be turned into a string.

In [ ]:

def integerify(string1, string2):

int1 = int(string1)

int2 = int(string2)

val\_sum = int1 + int2

val\_diff = int1 - int2

val\_prod = int1 \* int2

return val\_sum, val\_diff, val\_prod

integerify('20', '100')

Write a function that takes a list as the argument and returns a tuple consisting of two elements:

* A list with the items in reverse order.
* A list of the items in the original list that have an odd index.

**Instructor Note:** This function is a good example for introducing a couple of more advanced topics and could be written together with students versus walking them through it as it appears.

In [ ]:

def reverse\_and\_odd(input\_list):

reversed\_list = input\_list[::-1]

odd\_indices = []

for i in range(len(input\_list)):

if i % 2 == 1:

odd\_indices.append(input\_list[i])

return reversed\_list, odd\_indices

reverse\_and\_odd(names)

Quiz: Functions[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Quiz:-Functions)

Can you tackle these two challenges on your own?

*Write a function that takes a word as an argument and returns the number of vowels in the word.*

*Write a function that takes in a list of animals. Have it print out each animal's name in FULL CAPITAL LETTERS.*

* **Note:** You may need to do some outside research to find out how Python can capitalize all letters in a string!

Recaps and Requests[¶](http://localhost:8888/nbconvert/html/Desktop/SCHOOL_WORK/Week%201/welcome-to-data-science-master/intro_to_python_fundamentals.ipynb?download=false#Recaps-and-Requests)

Take a moment to write down the answers to the following for yourself:

1) What parts of the Python material covered today do I feel like I know very well right now?   
2) What parts of the Python material covered today were a struggle?

We'll each share what caused us some trouble today and take a few minutes to review anything that's outstanding. If you noticed that you really mastered something that somebody else found especially challenging, take some time to reach out and offer some help!