Name: Allan Gongora

Section: 0131

Lab 2a: Data Types

Welcome to Lab 2a!

Last time, we had our first look at Python and Jupyter notebooks. So far, we've only used Python to manipulate numbers. There's a lot more to life than numbers, so Python lets us represent many other types of data in programs.

In this lab, you'll first see how to represent and manipulate another fundamental type of data: text. A piece of text is called a *string* in Python.

You'll also see how to invoke *methods*. A method is very similar to a function. Calling a method looks different because the method is tied to a particular piece of data.

Last, you'll learn more about working with datasets in Python.

Please complete this notebook by filling in the cells provided. Before you begin, execute the following cell to load the provided tests.

In [1]:
```
pip install gofer-grader
```

```
Requirement already satisfied: gofer-grader in /opt/conda/lib/python3.7/site-packages
(1.1.0)
Requirement already satisfied: pygments in /opt/conda/lib/python3.7/site-packages (fr
om gofer-grader) (2.11.2)
Requirement already satisfied: jinja2 in /opt/conda/lib/python3.7/site-packages (from
gofer-grader) (3.0.3)
Requirement already satisfied: tornado in /opt/conda/lib/python3.7/site-packages (fro
m gofer-grader) (6.1)
Requirement already satisfied: MarkupSafe>=2.0 in /opt/conda/lib/python3.7/site-packa
ges (from jinja2->gofer-grader) (2.0.1)
Note: you may need to restart the kernel to use updated packages.
```

In [2]:
```
from gofer.ok import check
```

Recommended Reading:

- Data Types

1) For all problems that you must write explanations and sentences for, you **must** provide your answer in the designated space. This can include: A) Sentence reponses to questions that ask for an explanation B) Numeric responses to multiple choice questions C) Programming code

2) Moreover, throughout this lab and all future ones, please be sure to not re-assign variables throughout the notebook! For example, if you use `max_temperature` in your answer to one

question, do not reassign it later on. Otherwise, you will fail tests that you thought you were passing previously!

Once you're finished, select "Save and Checkpoint" in the File menu. Your name and course section number should be in the first and last cell of the assignment. Be sure you have run all cells with code and that the output from that is showing. Then click "Print Preview" in the File menu. Print a copy from there in pdf format. (This means you right click and choose print and choose "save as pdf" from your printer options.) You will need to submit the pdf in Canvas by the deadline.

The gopher grader output and/or output from your coding are essential to helping your instructor grade your work correctly and in a timely manner.

Files submitted that are missing the required output will lose some to all points so double check your pdf before submitting.

1. Review: The Building Blocks of Python Code

The two building blocks of Python code are *expressions* and *statements*. An **expression** is a piece of code that

- is self-contained, meaning it would make sense to write it on a line by itself, and
- usually has a value.

Here are two expressions that both evaluate to 3

```
3
5 - 2
```

One important form of an expression is the **call expression**, which first names a function and then describes its arguments. The function returns some value, based on its arguments. Some important mathematical functions are

Function	Description
`abs`	Returns the absolute value of its argument
`max`	Returns the maximum of all its arguments
`min`	Returns the minimum of all its arguments
`pow`	Raises its first argument to the power of its second argument
`round`	Round its argument to the nearest integer

Here are two call expressions that both evaluate to 3

```
abs(2 - 5)
max(round(2.8), min(pow(2, 10), -1 * pow(2, 10)))
```

All these expressions but the first are **compound expressions**, meaning that they are actually combinations of several smaller expressions. `2 + 3` combines the expressions `2` and `3` by addition. In this case, `2` and `3` are called **subexpressions** because they're expressions that are part of a larger expression. Any expression can be used as part of a larger expression.

A **statement** is a piece of code that *makes something happen* rather than *having a value*. For example, an **assignment statement** assigns a value to a name.

Every assignment statement has one `=` sign. The whole statement is executed by **evaluating the expression on the right-hand side** of the equals sign and then **assigning its value to the name on the left-hand side**. Here are some assignment statements:

```
height = 1.3
the_number_five = abs(-5)
absolute_height_difference = abs(height - 1.688)
```

A key idea in programming is that large, interesting things can be built by combining many simple, uninteresting things. The key to understanding a complicated piece of code is breaking it down into its simple components.

For example, a lot is going on in the last statement above, but it's really just a combination of a few things. This picture describes what's going on.

Explanation of the statement 'absolute_height_difference = abs(height - 1.688)'

Any names that you assign in one cell are available in later cells and can be used in place of the value assigned to them.

Question 1.1.
In the next cell, assign the name `new_year` to the larger number among the following two numbers:

1. the absolute value of $2^5 - 2^{11} - 2^1 - 2^0$, and
2. $5 \times 13 \times 31 + 6$.

Try to use just one statement (one line of code).

In [3]:
```
new_year = max(2**5 - 2**11 -2 - 1, 5 * abs(13 * 31) + 6)
new_year
```

Out[3]:
```
2021
```

Check your work by executing the next cell.

In [4]:
```
check('tests/q11.py')
```

Out[4]: All tests passed!

2. Text

Programming doesn't just concern numbers. Text is one of the most common types of values used in programs.

A snippet of text is represented by a **string value** in Python. The word "*string*" is a programming term for a sequence of characters. A string might contain a single character, a word, a sentence, or a whole book.

To distinguish text data from actual code, we demarcate strings by putting quotation marks around them. Single quotes (`'`) and double quotes (`"`) are both valid, but the types of opening and closing quotation marks must match. The contents can be any sequence of characters, including numbers and symbols.

We've seen strings before in `print` statements. Below, two different strings are passed as arguments to the `print` function.

In [5]:
```
print("I <3", 'Data Science')
```

```
I <3 Data Science
```

Just like names can be given to numbers, names can be given to string values. The names and strings aren't required to be similar in any way. Any name can be assigned to any string.

In [6]:
```
one = 'two'
plus = '*'
print(one, plus, one)
```

```
two * two
```

Question 2.1
Yuri Gagarin was the first person to travel through outer space. When he emerged from his capsule upon landing on Earth, he reportedly had the following conversation with a woman and girl who saw the landing:

```
The woman asked: "Can it be that you have come from outer space?"
Gagarin replied: "As a matter of fact, I have!"
```

The cell below contains unfinished code. Fill in the `...` s so that it prints out this conversation *exactly* as it appears above.

In [7]:
```
woman_asking = "The woman asked:"
woman_quote = '"Can it be that you have come from outer space?"'
gagarin_reply = 'Gagarin replied:'
gagarin_quote = "\"As a matter of fact, I have!\""

print(woman_asking, woman_quote)
print(gagarin_reply, gagarin_quote)
```

```
The woman asked: "Can it be that you have come from outer space?"
Gagarin replied: "As a matter of fact, I have!"
```

In [8]:
```
check('tests/q21.py')
```

Out[8]: All tests passed!

String Methods

Strings can be transformed using **methods**, which are functions that involve an existing string and some other arguments. One example is the `replace` method, which replaces all instances of some part of a string with some alternative.

A method is invoked on a string by placing a `.` after the string value, then the name of the method, and finally parentheses containing the arguments. Here's a sketch, where the `<` and `>` symbols aren't part of the syntax; they just mark the boundaries of sub-expressions.

```
<expression that evaluates to a string>.<method name>(<argument>,
<argument>, ...)
```

Try to predict the output of these examples, then execute them.

In [9]:
```
# Replace one letter
'Hello'.replace('H', 'C')
```

Out[9]:
```
'Cello'
```

In [10]:
```
# Replace a sequence of letters, which appears twice
'hitchhiker'.replace('hi', 'ma')
```

Out[10]:
```
'matchmaker'
```

Once a name is bound to a string value, methods can be invoked on that name as well. The name is still bound to the original string, so a new name is needed to capture the result.

In [11]:
```
sharp = 'edged'
hot = sharp.replace('ed', 'ma')
print('sharp:', sharp)
print('hot:', hot)
```

```
sharp: edged
hot: magma
```

You can call functions on the results of other functions. For example,

```
max(abs(-5), abs(3))
```

has value 5. Similarly, you can invoke methods on the results of other method (or function) calls.

In [12]:
```
# Calling replace on the output of another call to replace
'train'.replace('t', 'ing').replace('in', 'de')
```

Out[12]:
```
'degrade'
```

Here's a picture of how Python evaluates a "chained" method call like that:

In 'train'.replace('t', 'ing').replace('in', 'de'), 'train'.replace('t', 'ing')' is ran first and evaluates to 'ingrain'. Then 'ingrain'.replace('in', 'de') is evaluated to 'degrade'

Question 2.2

Assign strings to the names `you` and `this` so that the final expression evaluates to a 10-letter English word with three double letters in a row.

Hint: The call to `print` is there to print out the intermediate result called `the`. This should be an English word with two double letters in a row.

Hint 2: Run the tests if you're stuck. They'll give you some hints.

In [13]:
```
you = "keep"
this = "book"
a = 'beeper'
the = a.replace('p', you)
print('the:', the)
the.replace('bee', this)
```

```
the: beekeeper
```

Out[13]:
```
'bookkeeper'
```

In [14]:
```
check('tests/q211.py')
```

Out[14]: All tests passed!

Other string methods do not take any arguments at all, because the original string is all that's needed to compute the result. In these cases, parentheses are still needed, but there's nothing in between the parentheses. Here are some methods that take no arguments:

Method name	Value
`lower`	a lowercased version of the string
`upper`	an uppercased version of the string
`capitalize`	a version with the first letter capitalized
`title`	a version with the first letter of every word capitalized

In [15]:
```
'unIverSITy of caliFORnia'.title()
```

Out[15]:
```
'University Of California'
```

All these string methods are useful, but most programmers don't memorize their names or how to use them. Instead, people usually just search the internet for documentation and examples. A complete list of string methods appears in the Python language documentation. Stack Overflow has a huge database of answered questions that often demonstrate how to use these methods to achieve various ends.

Converting to and from Strings

Strings and numbers are different *types* of values, even when a string contains the digits of a number. For example, evaluating the following cell causes an error because an integer cannot be added to a string.

In [16]:
```
# 8 + "8"
```

However, there are built-in functions to convert numbers to strings and strings to numbers.

Function name	Effect	Example
`int`	Converts a string of digits and perhaps a negative sign to an integer (`int`) value	`int("42")`
`float`	Converts a string of digits and perhaps a negative sign and decimal point to a decimal (`float`) value	`float("4.2")`
`str`	Converts any value to a string (`str`) value	`str(42)`

Try to predict what the following cell will evaluate to, then evaluate it.

In [17]:
```
8 + int("8")
```

Out[17]:
```
16
```

Suppose you're writing a program that looks for dates in a text, and you want your program to find the amount of time that elapsed between two years it has identified. It doesn't make sense to subtract two texts, but you can first convert the text containing the years into numbers.

Question 2.3
Finish the code below to compute the number of years that elapsed between `one_year` and `another_year`. Don't just write the numbers `1618` and `1648` (or `30`); use a conversion function to turn the given text data into numbers.

In [18]:
```
# Some text data:
one_year = "1618"
another_year = "1648"

# Complete the next line.  Note that we can't just write:
#   another_year - one_year
# If you don't see why, try seeing what happens when you
# write that here.
difference = abs(int(one_year) - int(another_year))
difference
```

Out[18]:
```
30
```

In [19]:
```
check('tests/q221.py')
```

Out[19]: All tests passed!

Question 2.4

Use `replace` and `int` together to compute the difference between the year 753 BC (the founding of Rome) and the year 410 AD (the sack of Rome)). Try not to use any numbers in your solution, but instead manipulate the strings that are provided.

Hint: It's ok to be off by one year. In historical calendars, there is no year zero, but astronomical calendars do include year zero to simplify calculations.

In [20]:
```
founded = 'BC 753'
sacked = 'AD 410'
start = -int(founded.replace("BC ", ""))
end = int(sacked.replace("AD ", ""))
print('Ancient Rome lasted for about', end-start, 'years from', founded, 'to', sacked
```

```
Ancient Rome lasted for about 1163 years from BC 753 to AD 410
```

In [21]:
```
check('tests/q222.py')
```

Out[21]: All tests passed!

Strings as Function Arguments

String values, like numbers, can be arguments to functions and can be returned by functions. The function `len` takes a single string as its argument and returns the number of characters in the string: its **len**gth.

Note that it doesn't count *words*. `len("one small step for man")` is 22, not 5.

Question 2.5

Use `len` to find out the number of characters in the very long string in the next cell. (It's the first sentence of the English translation of the French Declaration of the Rights of Man.) The length of a string is the total number of characters in it, including things like spaces and punctuation. Assign `sentence_length` to that number.

In [22]:
```
a_very_long_sentence = "The representatives of the French people, organized as a Nati
sentence_length = len(a_very_long_sentence)
sentence_length
```

Out[22]:
```
896
```

In [23]:
```
check('tests/q231.py')
```

Out[23]: All tests passed!

3. Importing Code

> What has been will be again,
> what has been done will be done again;
> there is nothing new under the sun.

Most programming involves work that is very similar to work that has been done before. Since writing code is time consuming, it's good to rely on others' published codes when you can. Rather than copy-pasting, Python allows us to **import** other code, creating a **module** that contains all of the names created by that code.

Python includes many useful modules that are just an `import` away. We'll look at the `math` module as a first example. The `math` module is extremely useful in computing mathematical expressions in Python.

Suppose we want to very accurately compute the area of a circle with radius 5 meters. For that, we need the constant π, which is roughly 3.14. Conveniently, the `math` module has `pi` defined for us:

In [24]:
```
import math
radius = 5
area_of_circle = radius**2 * math.pi
area_of_circle
```

Out[24]:
```
78.53981633974483
```

`pi` is defined inside `math`, and the way that we access names that are inside modules is by writing the module's name, then a dot, then the name of the thing we want:

```
<module name>.<name>
```

In order to use a module at all, we must first write the statement `import <module name>`. That statement creates a module object with things like `pi` in it and then assigns the name `math` to that module. Above we have done that for `math`.

Question 3.1
`math` also provides the name `e` for the base of the natural logarithm, which is roughly 2.71. Compute $e^{\pi} - \pi$, giving it the name `near_twenty`.

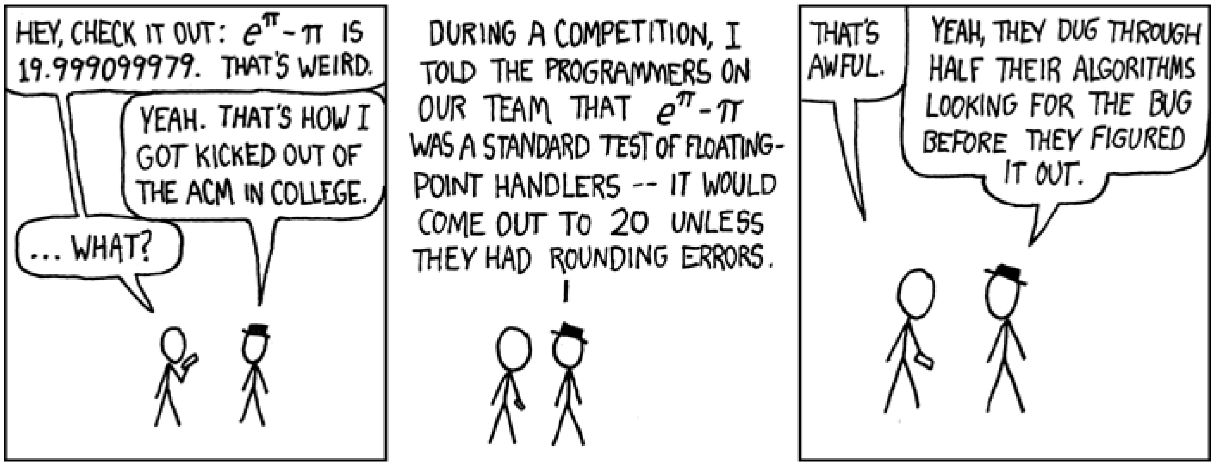
In [25]:
```
near_twenty = math.e**math.pi - math.pi
near_twenty
```

Out[25]:
```
19.99909997918947
```

In [26]:

```
check('tests/q31.py')
```

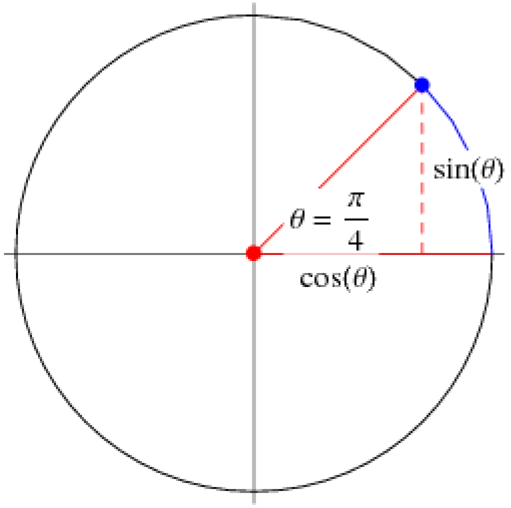
Out[26]: All tests passed!


HEY, CHECK IT OUT: $e^{\pi} - \pi$ IS 19.999099979. THAT'S WEIRD.
YEAH. THAT'S HOW I GOT KICKED OUT OF THE ACM IN COLLEGE.
... WHAT?
DURING A COMPETITION, I TOLD THE PROGRAMMERS ON OUR TEAM THAT $e^{\pi} - \pi$ WAS A STANDARD TEST OF FLOATING-POINT HANDLERS -- IT WOULD COME OUT TO 20 UNLESS THEY HAD ROUNDING ERRORS.
THAT'S AWFUL.
YEAH, THEY DUG THROUGH HALF THEIR ALGORITHMS LOOKING FOR THE BUG BEFORE THEY FIGURED IT OUT.

Importing Functions

Modules can provide other named things, including **functions**. For example, `math` provides the name `sin` for the sine function. Having imported `math` already, we can write `math.sin(3)` to compute the sine of 3. (Note that this sine function considers its argument to be in radians, not degrees. 180 degrees is equivalent to π radians.)

Question 3.2
A $\frac{\pi}{4}$-radian (45-degree) angle forms a right triangle with equal base and height, pictured below. If the hypotenuse (the radius of the circle in the picture) is 1, then the height is $\sin(\frac{\pi}{4})$. Compute that using `sin` and `pi` from the `math` module. Give the result the name `sine_of_pi_over_four`.



(Source: Wolfram MathWorld)

In [27]:
```
sine_of_pi_over_four = math.sin(math.pi / 4)
sine_of_pi_over_four
```

Out[27]:
```
0.7071067811865475
```

In [28]:
```
check('tests/q311.py')
```

Out[28]: All tests passed!

For your reference, here are some more examples of functions from the `math` module.

Note how different methods take in different number of arguments. Often, the documentation of the module will provide information on how many arguments is required for each method.

In [29]:
```
# Calculating factorials.
math.factorial(5)
```

Out[29]:
```
120
```

In [30]:
```
# Calculating logarithms (the logarithm of 8 in base 2).
# The result is 3 because 2 to the power of 3 is 8.
math.log(8, 2)
```

Out[30]:
```
3.0
```

In [31]:
```
# Calculating square roots.
math.sqrt(5)
```

Out[31]:
```
2.23606797749979
```

There's many variations of how we can import methods from outside sources. For example, we can import just a specific method from an outside source, we can rename a library we import, and we can import every single method from a whole library.

In [32]:
```
# Importing just cos and pi from math.
# Now, we don't have to use "math." before these names.
from math import cos, pi
print(cos(pi))
```

```
-1.0
```

In [33]:
```
# We can nickname math as something else, if we don't want to type the name math
import math as m
m.log(m.pi)
```

Out[33]:
```
1.1447298858494002
```

In [34]:
```
# Lastly, we can import ever thing from math and use all of its names without "math."
from math import *
log(pi)
```

Out[34]:
```
1.1447298858494002
```

A Function that Displays a Picture

People have written Python functions that do very cool and complicated things, like crawling web pages for data, transforming videos, or learning functions from data. Now that you can import things, when you want to do something with code, first check to see if someone else has done it for you.

Let's see an example of a function that's used for downloading and displaying pictures.

The module `IPython.display` provides a function called `Image`. The `Image` function takes a single argument, a string that is the URL of the image on the web. It returns an *image* value that this Jupyter notebook understands how to display. To display an image, make it the value of the last expression in a cell, just like you'd display a number or a string.

Question 3.3
In the next cell, import the module `IPython.display` and use its `Image` function to display the image at this URL:

```
https://upload.wikimedia.org/wikipedia/commons/thumb/8/8c/David_-
_The_Death_of_Socrates.jpg/1024px-David_-_The_Death_of_Socrates.jpg
```

Give the name `art` to the output of the call to `Image`. (It might take a few seconds to load the image. It's a painting called *The Death of Socrates* by Jacques-Louis David, depicting events from a philosophical text by Plato.)

Hint: A link isn't any special type of data type in Python. You can't just write a link into Python and expect it to work; you need to type the link in as a specific data type. Which one makes the most sense?

In [35]:

```
# Import the module IPython.display. Watch out for capitalization.
import IPython.display
# Replace the ... with a call to the Image function
# in the IPython.display module, which should produce
# a picture.
art = IPython.display.Image("https://upload.wikimedia.org/wikipedia/commons/thumb/8/8
art
```

Out[35]:

In [36]:

```
check('tests/q312.py')
```

Out[36]: All tests passed!

4. Submission

Once you're finished, select "Save and Checkpoint" in the File menu. Your name and course section number should be in the first and last cell of the assignment. Be sure you have run all cells with code and that the output from that is showing.

Double check that you have completed all of the free response questions as the autograder does NOT check that and YOU are responsible for knowing those questions are there and completing them as part of the grade for this lab.

When ready, click "Print Preview" in the File menu. Print a copy from there in pdf format. (This means you right click and choose print and choose "save as pdf" from your printer options.) You will need to submit the pdf in Canvas by the deadline.

The gopher grader output and/or output from your coding are essential to helping your instructor grade your work correctly and in a timely manner.

Files submitted that are missing the required output will lose some to all points so double check your pdf before submitting.

In [37]:

```
# For your convenience, you can run this cell to run all the tests at once!
import glob
```

```
from gofer.ok import grade_notebook
if not globals().get('__GOFER_GRADER__', False):
    display(grade_notebook('lab02a.ipynb', sorted(glob.glob('tests/q1*.py'))))
```

```
I <3 Data Science
two * two
The woman asked: "Can it be that you have come from outer space?"
Gagarin replied: "As a matter of fact, I have!"
sharp: edged
hot: magma
the: beekeeper
Ancient Rome lasted for about 1163 years from BC 753 to AD 410
-1.0
['tests/q11.py', 'tests/q21.py', 'tests/q211.py', 'tests/q221.py', 'tests/q222.py',
'tests/q231.py', 'tests/q31.py', 'tests/q311.py', 'tests/q312.py']
Question 1:
```

All tests passed!

```
Question 2:
```

All tests passed!

```
Question 3:
```

All tests passed!

```
Question 4:
```

All tests passed!

```
Question 5:
```

All tests passed!

```
Question 6:
```

All tests passed!

```
Question 7:
```

All tests passed!

```
Question 8:
```

All tests passed!

```
Question 9:
```

All tests passed!

```
1.0
```

Name: Allan Gongora

Section: 0131