Python Session 1

Feel free to use the outline on the left side of google docs to quickly jump through the document.
After going through the content, please take a minute to respond to this feedback survey of our reference documents.

Variables

Variables are used to store data. Data can be integers, floats (decimal numbers), strings, or booleans. A variable can only store *one* type of data at a time and is created (dynamically casted) when it is first assigned a value.

```
a = 5                # the variable a is created and assigned the integer 5
1a = 6.6             # this is an invalid variable name
```

A variable's data type is *not permanent*. A variable can change its data type to fit its assigned data at any time, which is called **dynamic casting**.

```
a = 5                # variable a is an integer storing the number 5
a = "hello world"    # variable a is now a string storing "hello world"
```

Strings

Strings are text surrounded by either single quotes or double quotes.

⚠ The quote type used to surround the string cannot be used as a character in the string without being an escape character.

```
a = "Closing with quotes creates a string"   # this string is surrounded by double quotes

a = "Scott said \"hello\" to John's dog" # \" allows us to use " in string surrounded with "
a = 'Scott said "hello" to John\'s dog'  # \' allows us to use ' in string surrounded with '
```

In the example above, we used single quotes to surround a string. Therefore, we can use double quotes inside the string without escape characters while the single quote in "John's" requires an escape character. The opposite situation would also work.

Python Standard Template Library (STL)

Printing - print()

`print(...)` takes in any number of **strings/string variables** and outputs them onto the screen/terminal followed by a new line. The **strings/string variables** passed in can be separated by either commas or + (concatenation).

- **Comma** joins strings and creates spaces in between each string
- **+** combines strings, but does not create a space between each string

`print()` can also take in other arguments (what we pass in):

- `end=` allows us to specify how to end the printed string
- `sep=` allows us to specify what to put in between each *comma-separated* string passed in

```
w = "World"
print("Hello", w, end="!")  # end="!" will overwrite the ending: it won't create a new line
print(w, w, sep="-")        # sep="-" will place a - instead of a space between each string
```

```
Hello World!World-World
```

⚠ Be aware that `sep=` will not have an effect on concatenated strings.

Input - input()

`input(...)` fetches an input from the user/keyboard/terminal. Inputs are stored as a string. A string prompt can be passed in to ask the user for inputs. Prompts do ***not*** end with a new line, but one can be added using the *end line escape character* "`\n`".

```
x = input("Input: ")     # Asks the user for input after the prompt
print(x)                 # Prints the input
```

```
Input: Sahas is Cool
Sahas is Cool
```

Typecasting

Non-strings (i.e. integers) cannot be concatenated (+) to strings . Attempting to concatenate a string with a non-string will result in an error message. This is especially important when passing in concatenated strings to functions like **`print()`** and **`input()`**.

In addition, the *return type* (the fetched input) for **`input()`** is a string. Many times, we may want to store the result of the input as a different data type.

We can solve both of the above problems by **typecasting** a variable/literal into whatever data type we want using these functions: **`int(...)`** to cast into an integer, **`str(...)`** to cast into a string, or **`float(...)`** to cast into a float.

```
x = int(input("Input:" + " "))
print("First Type:", x)           # x here is still an integer, but a comma is used
print("Second Type: " + str(x))   # Strings are concatenated, so x must be casted to a string
```

```
Input: -2
First Type: -2
Second Type: -2
```

Conditional Statements

Booleans Expressions

Booleans are data types that are either **`True`** or **`False`**. **`True`** can also be represented by integer **`1`** and **`False`** with **`0`**.

Boolean expressions are statements made of comparisons that result in either **`True`** or **`False`**. Boolean expressions follow an order of operations similar to math.

Comparisons can be made using the following **comparison operators**:

- **`==`** checks for **equality**
- **`< or >`** checks for **less than** or **greater than**
- **`<= or >=`** checks for **less than or equal to** or **greater than or equal to**.

Multiple expressions and/or booleans can be a combined using **logical operators**:

- **`and`** requires **both** booleans expressions to be **`True`**. This is **`&&`** in *C*.
- **`or`** requires **at least one** of the boolean expressions to be **`True`**. This is **`||`** in *C*.
- **`not`** **inverts** the result of the boolean expression. This is **`!`** in *C*.

```
x = False
y = True
print(x and y)              # x and y must both be True
print(x or not y)           # either x or the opposite of y must be True
```

```
False
False
```

If, Elif, Else

Conditional statements use boolean expressions as conditions to determine whether to run a certain piece of code. The three conditional keywords are **`if, elif, and else`**. A group of **`if, elif, and else`** forms a conditional code block.

The keywords **`if`** and **`elif`** require boolean expressions after. There can be zero or more **`elif`** statements following an **`if`**.

For each branch of the block, that branch is executed only if its condition evaluates to **`True`** with the exception of the **`else`**. The **`else`** statement is optional and only evaluated when all other branches of the block evaluate to **`False.`**

```
x = False
if not x:           # not x is the same as saying the opposite of False, which is True
    print(x)        # a singular additional ident tells us this line is part of the if branch
```

```
False
```

`input()` allows the user's input to determine which branch is entered.

```
x = input("Input: ")
if not (x == "PYTHON" or x == "python"):
    print("Use \"PYTHON\" instead!")
elif x == "Ruby":
    print("Close")
else:
    print(x)
```

```
Input: PYTHON
PYTHON
```

```
Input: JavaScript
Use "PYTHON" instead!
```

⚠ After the colon, there **MUST** be some code, otherwise an error message will be given.

⚠ Branches are evaluated top to bottom. If a branch is evaluated, the rest of the block **is not** evaluated.

Loops

While and For

Loops allow you to repeat instructions multiple times. There are two types of loops: **`while`** and **`for`**.

`while` loops are useful when the number of repeats is unknown. Boolean expressions determine whether to loop.

`for` loops are used when the number of repeats is known and often iterate through something.

```
x = 1
while not x >= 4:   # loops while i is less than 4
    print(x)
    x *= 2

# range(start, end, increment_amount). Note that we don't have to pass in all of the arguments
for i in range(0, 6, 3):   # i starts at 0 and increase by 3 while (i < 6)
    print(i)
```

⚠ After the colon, there **MUST** be some code, otherwise an error message will be given.

Flow Control

Flow controls control the flow of the loops:

- `break` - exit the loop it is inside of
- `continue` - immediately starts the next iteration of the loop
- `pass` - does nothing; placeholder

```
x = 3
while True:
    if x == 3:
        break
```

```
while True:
    continue
    print("Hey!")
    # print never runs
```

```
while True:
    pass
    # nothing happens
    # loop keeps going
```

Library: random

Libraries are your best friends. They contain pre-written code that can be imported with the keyword `import` with the library name following it. Usually you import libraries at the **top** of the program so the entire program has access. You can also add `... as [custom_alias]` after the import if you wish to use an alias rather than the full library name when using its functions. **Most libraries have a preferred alias that is recognized by other programmers**. We will see more of these aliases later (particularly in Phase 2).
All usages of a library's functions follow this format: `[library_name].[function]`.
The library we will be using is `random` (alias `rand`):

- `randint(a, b)` is a function that generates a random integer between a and b *inclusive*.

```
import random as rand  # imports the random library with the rand alias

rand.randint(0, 5)     # [library_name].[function]
```

Extras

Escape Characters

Escape characters are used in strings to perform a special string property. Escape characters are written with a **backslash** \ followed by **some character**.
During Session 1, we see `\n`, `\'` and `\"` being used. This is only two of many escape characters:

Esc Char	Purpose	Notes
\'	Single Quote i.e. 'Sarah\'s cake'	These are used so the quote doesn't end the string when we don't want to
\"	Double Quote i.e. "He said \"Hello\"."	
\\	Backslash w/o creating an escape char	
\n	Creates a new line	This is probably the most useful
\t	Creates an indentation in the string	
\b	Deletes the previous char	This is redundant in most cases

Error Messages

Error messages are a normal result of programming. Errors tell you which specific line in your code may have an error.

```
print("Good Morning!")
print("It is" + 3 + " thirty")
print("Good Night!")
```

```
Good Morning!
Traceback (most recent call last):
  File "main.py", line 2, in <module>
    print("It is" + 3 + " thirty")
TypeError: can only concatenate str (not "int") to str
```

Be aware that error messages **do not** always tell you the exact line where the error is, but it may indicate that the error is around the specified location. If the provided location looks correct, it is likely the error may be on the previous or next line(s).

```
print("Good Morning!")
for i in range(0, 5):
                          # error actually occurs here
print("Good Night!")
```

```
  File "main.py", line 4
    print("Good Night!")
    ^
IndentationError: expected an indented block
```

Order of Operations

Math and boolean expressions follow an order of operations/evaluation. This is similar to PEMDAS in math with the addition of a few additional coding syntaxes. Ordering from highest to lowest precedence:

1. Parentheses ()
2. Exponents **
3. Unaries -x
4. Mult/Div/Mod */%
5. Add/Sub +-
6. comparison ops ==/</>/<=/>=
7. not
8. and/or

Testing

A key component in computer science is testing. It is important to test your code incrementally to make sure your program runs and does what it is supposed to do. Testing can include: **testing different inputs, comparing outputs, or simple debugging** of different parts of the program.
When testing inputs, it is important that you also **test extreme/edge cases**. *For example, in a calculator program, an edge case may be dividing by zero, or multiplying past the integer limit, etc.* Oftentimes, **edge cases can fail** the program **while regular cases may not**, so **always** test extreme cases to make sure your program works correctly.

Linux/Terminal Commands

Programmers often use linux/terminal commands to perform certain operations like moving through directories, or creating files, etc. For this Python course, the most we will see are these two commands:

- **`cd <directory>`** = move into the provided directory. **`..`** will move out of the current directory
- **`ls`** = List the contents in this directory

Main

Since python is a scripting language, the code runs sequentially and there is no actual **`main()`** function. However, you can still create a "**`main()`**" function if you wish by adding this in your code:

```
if __name__ == "__main__":

```

Be aware that this behaves differently from the **`main()`** from languages like *C++* that we will not be covering in this Python course.