Lab 2 - Python Basics

Last week, we created a "static" experiment using PsychoPy. Today, we will learn how to embed Python code in our experiment to randomize and control which images we'll be displaying in each trial. Before we return to the PsychoPy experiment, let's cover some basic concepts in Python.

My goal is to get through this worksheet quickly, and then returning to the last portion of Lab 1, where we use some of our knew Python knowledge in "code components".

Topics to be covered:

- syntax (significant spacing; assigning variables; commenting code)
- data types (string, int, bools, float)
- data containers (lists and dictionaries)
- control flow (`if..elif..else` statements)
- loops (`for` and `while` loops)
- functions
- useful functions
- object-oriented programming
- intro to PsychoPy GUI

Python Syntax

Significant Spacing (Indentation)

Unlike other programming langauges, how you choose to indent your code is very important in Python. This is because indents tell the Python interpreter which chunks of code should be ran together before moving onto the next part of the program.

If you fail to indent things properly, you may get a syntax error. For instance, consider the block of code below, which will print the statement "Yes, 1 is greater than 0" if the expression 1 > 0 returns true. Note the indent following the new line.

In [ ]:
```
if 1 > 0:
    print('Yes, 1 is greater than 0')
```

By indenting, we indicate the two seperate code blocks needing to be executing: (1) evaluating the expression 1 > 0 and (2) printing the string 'Yes, 1 is greater than 0'. In general, indentation signfies each individual code block (i.e., a set of commands that should be executed together before moving onto the next part of the program).

Whenever we type a `:` in Python, the interpreter will expect that you indent the next line of code. Notice, what happens if I fail to indent the second line of code, an error is returned.

In [ ]:
```
if 1 > 0:
print('Yes, 1 is greater than 0')
```

In [ ]:
```
#fix the error in this code:
if 1 > 2:
```

```
print("1 is greater than 2")
else:
print("2 is greater than 1")
```

Defining variables

Variables are convenient ways for us to store values that we may want to use later in our program. We *assign* a value to a variable by using the `=` operator. We can later *access* this variable by simply calling the name of the variable.

In [ ]:
```
# assigning a variable called a:
a = 24
```

In [ ]:
```
# accessing a variable named a:
print(a)
```

In [ ]:
```
# using variable a:
a + 14
```

There are some restrictions for what you can use as your variable names. For instance, invalid variable names have spaces and lead with number. See the examples below.

In [ ]:
```
# leading with numbers are not okay
2name = 'some string'

# numbers are okay, you just can't lead with them
name2 = 'some string'

#no special characters like *, !, @, #, ect
name2* = 'some string'

# no spaces
na me2 = 'some string'
```

Comments

It is good practice to comment your code. Commenting means that you provide a verbal description of your code, making sure that you and others will remember what your code is supposed to do. For best practices on commenting your code, check out this website: https://www.elegantthemes.com/blog/wordpress/how-to-comment-your-code-like-a-pro-best-practices-and-good-habits

There are two ways to comment your code:

```
With the pound sign #
Between a pair triple quotes ''' '''
```

When you use these symbols, you tell the python interpreter to ignore whatever text follows.

In [ ]:
```
#This is a comment using the pound sign. Note that you can only comment out one line of te
```

In [ ]:
```
"""
```

```
This is a multi-line comment.
You can write more lengthy descriptions in this
type of comment.
"""
```

Basic data types

In this section I will go over the basic data types in Python and how you use them.

The basic data types are:

```
strings (more info: https://www.w3schools.com/python/python_strings.asp)
ints (more info: https://www.tutorialspoint.com/python/python_numbers.htm)
booleans (or 'bools') (more info:
https://www.w3schools.com/python/python_booleans.asp)
floats and doubles
```

If you would like to go deeper into data types than what I cover here, check out these resources:

```
https://docs.python.org/3/tutorial/datastructures.html
http://www.math.wsu.edu/math/kcooper/M300/pythontypes.php#:~:text=These%20are%20almost%20
```

Strings

You can define a string with either single quote `''` or double quotes `""` . A string variable will then represent the characters you placed between the quotes. Below are a few examples of strings.

In [ ]:
```
string1 = 'i am a string'
string2 = '2352&(#)'
string3 = "35"
notstring = 35
```

In [ ]:
```
#We can check if these variables are strings by using the `type()` function.
type(string1)
```

In [ ]:
```
#We can test if a variable is a string. Output is a boolean (True/False):
type(string2) == str
```

In [ ]:
```
#Another way to test if a variable is a string. Output is a boolean (True/False):
isinstance(string3,str)
```

In [ ]:
```
#We can include boolean statements in if statements:
if isinstance(notstring,str):
    print("This variable is a string")
else:
    print("This variable is not a string")
```

In [ ]:
```
#you can find the length of a string by using the following function. note that len will
#characters in the string
len(string1)
```

Integers

Integers are the most basic type of numeric representation in Python. Below are a few examples of integers (or 'ints')

In [ ]:
```
int1 = 3
int2 = 243242
int3 = 300

# how do you check if variable int1 is an integer?
```

Some math operations on integers:

In [ ]:
```
int1 - 2
```

In [ ]:
```
#Divison with a backslash
3/2
```

In [ ]:
```
#multplication is performed using a *
123*3
```

In [ ]:
```
#take exponents using **
3**2
```

In [ ]:
```
#greater than is with a >
3 > 2
```

In [ ]:
```
#greater than and equal to is >=
3 >= 3
```

In [ ]:
```
#you can create more complex expression using parens ()
(3 + 2) / 2 ** 2 > (34 * 2)/4
```

Exercise

Can you explain why these two expressions are different? What may be going on?

In [ ]:
```
value1 = 9 / 2

print(value1)
```

In [ ]:
```
value2 = 10 // 3

print(value2)
```

Exercise

The modulus operator is very common in computer programming. Given the examples below, can you figure out what the modulus operation `%` is doing?

In [ ]:
```
7 % 2
```

In [ ]:
```
8 % 2
```

In [ ]:
```
10 % 3
```

Floats and doubles

Floats and doubles are basically numbers with decimal points. For our present purposes, we wont worry too much about the difference between floats and doubles (if you are interested, read this: https://hackr.io/blog/float-vs-double), but just keep in mind that if you want to represent a number to more precision than just its integer value (i.e., include decimal points), the resulting data type is no longer an `int` but rather a `float` or `double` . See the examples below:

In [ ]:
```
type(3.5)
```

In [ ]:
```
a = 9
b = 2

c = a/b

print(c)
print(type(a))
print(type(b))
print(type(c))
```

Booleans

It is oftentimes important to know if an expression is true or false. Booleans (or 'bools') are data types that return either a `True` or `False` value given the truth value of an expression.

I will give a few examples for how you can work with and assign bool values.

In [ ]:
```
bool1 = True

print(bool1)
```

In [ ]:
```
bool2 = 3 + 3 < 5

print(bool2)
```

In [ ]:
```
bool3 = 17 == 17

print(bool3)
```

In [ ]:
```
bool4 = bool3 == bool2

print(bool4)
```

In [ ]:

```
s = 'this is a string'
bool5 = type(s) is str
print(bool5)
```

Difference between = and ==

What is going on with this statement `bool4 = bool3 == bool2` ?

You may now notice that there is a difference between = and ==. Remember, = is for *variable assignment* and == is for evaluating whether two values are equivalent.

Lists

Lists are defined by using square brackets `[]` and will contain a list of other data types. These other data types can be ints, strings, other lists, among other things.

(more info: https://www.w3schools.com/python/python_lists.asp)

In [ ]:

```
list1 = [1, 4, 6]
print(list1)
```

In [ ]:

```
list2 = ['apple', 'orange', 'pear']
print(list2)
```

Oftentimes, you'll want to interact with or modify your lists. You can do so using a variety approaches, including built-in methods. A review of Python list methods can be found here: https://www.programiz.com/python-programming/methods/list/index. We will go over common use cases of lists.

Accessing an element in the list

Importantly, in Python, the first index of a list starts with `0`. This is different from some other languages like Matlab, which starts with a `1`.

In [ ]:

```
item = list2[1]

print(item)
```

You may also access arange of elements using the following notation:

In [ ]:

```
#here we access the 2nd to 7th elements in a list and assign it to a new list variable
big_list = ['orange','apple','pear','watermelon','cherry','lime','lemon', 3, 6, ['list', '
small_list = big_list[2:7]
print(small_list)
```

Replacing an element in the list

In [ ]:

```
list2[0] = 3

print(list2)
```

Adding an element to the list

There are two main ways to add an element to the list.

Using the `append()` method:

In [ ]:
```
print(list1)

list1.append(3)

print(list1)
```

Using the `+` operator, you can concatenate two lists together. You can also add a single element if it is contained within a list:

In [ ]:
```
# adding a single element:
list1 = list1 + [3]
print(list1)

# concatenating two lists:
list_concatenated = list1 + list2
print(list_concatenated)
```

Exercise 2.1

(a) Fix the error in this code so that it adds an element to the list:

In [ ]:
```
list1 = list1 + 8

print(list1)
```

(b) Does the following code concatenate `list1` and `list2` ?

In [ ]:
```
list1.append(list2)

print(list1)
```

(c) If you want to use a list method instead of the `+` operator to concatenate two lists, you can use `extend()` . Use the `extend()` list method to concatenate `list1` and `list2` .

In [ ]:
```
# Concatenate list1 and list2 using extend():
```

Removing an element (if you know the index)

Use the `pop()` method:

In [ ]:
```
# Remove the 2nd item from the list:
print(big_list)
big_list.pop(1)
print(big_list)

# Remove the last item from the list:
big_list.pop(-1)
print(big_list)
```

Use the `del` statement:

In [ ]:
```
# delete the 1st and 2nd element of the list:
```

```
del big_list[1]
print(big_list)
```

Copying a list

Let's say you want to modify a list, but you want to save the state of the current list. How would you go about doing so?

In Matlab, you might assign an array (or cell or data struct) to a different variable and modify the new array variable. What happens if you use this approach in Python?

In [ ]:
```
# Matlab-like approach:
fruit_list = ['orange','apple','pear','watermelon','cherry','lime','lemon']
print(fruit_list)

# Assign list to a new variable:
temp_list = fruit_list

# Modify temp_list:
temp_list[0] = 'dragonfruit'
temp_list.pop(-1)

# Print temp_list
print(temp_list)
```

Everything looks fine right? But what happens when we print `fruit_list`, which is the original copy that we wanted to save:

In [ ]:
```
print(fruit_list)
```

As you can see, the modifications that we did to `temp_list` transfers to `fruit_list`. When we make assignments in Python, we did not actually make a copy of the object. In order to make a copy, we will use the `copy()` method:

In [ ]:
```
fruit_list = ['orange','apple','pear','watermelon','cherry','lime','lemon']
print(fruit_list)

# Assign a copy of fruit_list to temp_list
temp_list = fruit_list.copy()

# Modify temp_list:
temp_list[0] = 'dragonfruit'
temp_list.pop(-1)

# Print temp_list
print(temp_list)

# Print the original fruit_list:
print(fruit_list)
```

Find the length of a list

You can find the length of a list (how many elements are contained in it) by using the `len()` function:

In [ ]:
```
len(fruit_list)
```

Exercise 2.2

Use `len()` function to find and print the lengths of `list1` and `list2`. Why do you think `list2` is longer than `list1`?

In [ ]:
```
list1 = [1, 2, 3, 4, 5]
list2 = [1, 2, 3, [4, 5, 6]]

list3 = [1,'2']
```

Dictionaries

Dictionaries are similar to lists, but they assign a key-value pair to represent items. Dicts are defined using curly braces `{}` and must be provided at least one key-value pair, which takes the following form `key:value`. While values can be whatever data type you like (lists, integers, strings, other dictionaries), key's must be a string.

(more info: https://www.w3schools.com/python/python_dictionaries.asp)

In [ ]:
```
shopping_list = {
    'fruits': ['orange', 'apple','pear'],
    'soda': ['coke','dr. pepper'],
    'vegetables': None
}
shopping_list
```

In [ ]:
```
#we access values in a dict by indexing the dictionary with the key name
shopping_list['fruits']
```

In [ ]:
```
#we can easily add to our dictionary in one of two ways. First we can add an item to a key
shopping_list['fruits'].append('banana')

print(shopping_list['fruits'])
```

In [ ]:
```
#or we can create a new key and add it to the dictionary
shopping_list['breads'] = 'Daves Killer Bread'
shopping_list
```

Exercise 2.3

Why do you think we can use the `append()` method with `shopping_list['fruits']`? What data type is `shopping_list['fruits']`? Use the `type()` function to find out the data type of `shopping_list['fruits']` and `shopping_list`:

In [ ]:
```
# Use type() to figure out the data types:

type(shopping_list['fruits'])
```

Exercise 2.4

Below are two dictionaries - `subject_data` contains the data for a single subject and `experiment_data` contains the compiled dataset of subjects 1 thru 3. Add subject 4's data to the `experimend_data` dictionary. There are a couple of ways to accomplish this - try to use the most programmatic approach.

In [ ]:
```
# Subject 4's data
subject_data = {
    'subjectID': [4],
    'age': [21],
    'rt': [43.1],
    'condition': [2]
}

# Compiled data
experiment_data = {
    'subjectID': [1, 2, 3],
    'age': [23, 25, 20],
    'rt': [34.44, 24.2, 55.3],
    'condition': [1, 2, 1]
}


# Add sub 4's data to the compiled dataset dictionary:
```

Control flows

You may want to branch your code so it performs different operations given the truth values of other expressions. To do so you can use an if-else statement. The syntax follows:

In [ ]:
```
if expression:
    # Do something
elif expression:
    # Do something else
else expression:
    # Do something else
```

Note that the `elif` expression is optional, and you can just use a simple `if..else` statement.

For more detailed information, see: https://www.w3schools.com/python/python_conditions.asp

Let's look at a few examples to make this idea more concrete.

In [ ]:
```
expression = 1 == 1

if expression:
    print('The expression evaluated as true!')
else:
    print('The expression evaluated as :(')
```

In [ ]:
```
expression = 1 == 2

if expression:
    print('The expression evaluated as true!')
else:
    print('The expression evaluated as :(')
```

In [ ]:
```
list1 = ['apples','pear','orange']

if len(list1[0]) < 1:
    print(list1[0])
else:
    #what do you think the syntax 1: is telling python to do?
    print(list1[1:])
```

Loops

We will now go over the basic ways to iterate over data types like lists and dicts. If you would like more information, check out this website https://www.w3schools.com/python/python_for_loops.asp

We will focus on two types of loops the `for` loop and the `while` loop.

For loop

The for loop will iterate over a list of items (either a list a dictionary or a set). When starting out, you will be using for loops more than while loops (although both do appear). I will provide a few examples of how you can use for loops.

Here's an example to get an idea of the `for` loop syntax:

In [ ]:
```
some_iterable_data = range(0,3)
for variable_name in some_iterable_data:
    print(variable_name)
```

You can put whatever label you want for `variable_name` , but the data type in `some_iterable_data` must be a data type that you can iterate over. You will then perform a series of computations with each item you pulled form the object you are iterating over.

I will provide a few examlpes to make this clearer. This first example uses a list, which is an iterable data type.

In [ ]:
```
list1 = ['a', 2 , [1, 2, 3]]

for item in list1:
    print(item)
```

In [ ]:
```
#notice how my switching the label item to somethign else, the loop will still compute.
for some_other_label in list1:
    print(some_other_label)
```

In [ ]:
```
list1 = ['a', 2 , [1, 2, 3]]

for item in list1:
    if type(item) == list:
        print(item)
```

In [ ]:
```
list1 = ['a', 2 , 3, 4, 5, [1, 2, 3]]
new_list = []

for item in list1:
    if type(item) == int:
        new_list.append(item)

print(new_list)
```

In [ ]:
```
list1 = ['apple','orange','pear','one', 1, 'two', 2]

for item in list1:
    #instead of saying == to see if two variables equal one another, we can say is
    if type(item) is str:
        print('%s is a string' % item)
    elif type(item) is int:
        print('%d is an int' % item)
```

While loop

The while loop will continue to run until an expression returns `false`.

The program will continue to execute the code in the block repeatedly until the expression after the `while` statement evalues to False. Here's an example:

In [ ]:
```
count = 0

while count < 100:
    count = count + 10
    print('Count is now at %d' % count)
```

Functions

We use the `def` statement to define a function. For example, the function `concatenate` below takes two arguments `string1` and `string2` and will return a concatenated string separated by a space. Note that the `return` statement specifies the output of the function:

In [ ]:
```
def concatenate(string1, string2):
    return string1 + ' ' + string2
```

In [ ]:
```
concatenate('Hello,', 'Class!')
```

What happens if we pass more than two arguments?

In [ ]:
```
concatenate('Hello,','it''s','me...')
```

Exercise 2.5

Let's create a more robust concatenate function that handles a variable number of strings. This function should accept **one list argument**, where the list can contain any number of strings. Hint - `for` loops and `len()`:

In [ ]:
```
# This is the test input argumetn:
test_argument = ['Try','to','concatenate','this','list','of','strings']
```

Object Oriented Programming

Python implements a style of coding called Object Oriented Programming. In a nutshell, Python allows you to create **classes**, which are structures that contain **methods** and **attributes**. Methods and attributes are like functions and variables that are associated with the class.

Let's look at an example defining a `class`:

In [ ]:
```
class Dog:

    # Class Attribute
    species = "Canis familiaris"

    def __init__(self, name, age):
        # Instance Attributes
        self.name = name
        self.age = age

    # Instance Method
    def bark(self):
        return 'woof'

    # Another instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"

    def introduce(self):
        return f"My name is {self.name}. I am {self.age}. I am a {self.species}. Nice to r
```

In order to do anything with the class, we need to **instantiate an object** of the class like so:

In [ ]:
```
# Instantiate a Dog object
molly = Dog('Molly', 13)
```

You have instantiated `molly` , an object of the `Dog` class, whose name is 'Molly' and age is 13.

You can call methods and modify attributes using **dot notation**. You're already familiar with dot notation from using them with lists. Lists are a built-in Python class, and `append()` , `pop()` etc. are just different methods of the List class!

You can call one of `molly` 's methods:

In [ ]:
```
molly.introduce()
```

In [ ]:
```
molly.speak('bow wow')
```

You can update the attributes of `molly` :

In [ ]:
```
# Change Molly's age from 13 to 14:
molly.age = 14

# Change Molly's species:
molly.species = 'Good Girl'

# Introduce Molly:
molly.introduce()
```

OOP is widely used when you're programming with PsychoPy (in fact, all the components in the GUI are classes!).

If you would like to learn more about object-oriented programming, check out this tutorial: https://realpython.com/python3-object-oriented-programming/