Workshop Notebook 1: Introduction to Jupyter & Python

Mandatory Disclosures

1. This is a whirlwind introduction, not exhaustive instruction
2. All images are by courtesy of the University Archives at Texas State University: http://www.univarchives.txstate.edu (http://www.univarchives.txstate.edu)
3. img_qc_workshop is licensed under the GNU General Public License v3.0, https://github.com/photosbyjeremy/img_qc_workshop/blob/master/LICENSE (https://github.com/photosbyjeremy/img_qc_workshop/blob/master/LICENSE)
4. *Any and all code provided is done so without any warranty or expectation of support by Jeremy Moore, Todd Peters, or Texas State University*

Jupyter Notebooks

Click on *Help* > *User Interface Tour* in the menu for a quick introduction to the UI

- What you HAVE to know:
    - Cells can contain Markdown, code, or raw text
    - Click on a cell to select it and click again to modify its content
    - **Shift+Enter to execute a cell and select the next one** (Ctrl+Enter if you do not want to select the next one)

- What's helpful to know:
    - You can copy/paste and insert/remove cell using the menu above
    - **Keyboard Shortcuts!!**
        - a: add cell before selected cell
        - b: add cell after selected cell
        - dd: delete selected cell

```
In [ ]:  # This is a code cell.

         # Comments can be entered in a code cell after the pound-sign
         x = 'variable'
         print(f'x is a {x}')
```

This is a Markdown cell.

"Markdown is a text-to-HTML conversion tool for web writers." -- https://daringfirebal.net/projects/markdown/

This is Raw NBConvert Cell. Write raw LaTeX or other formats here. "Running" this cell does nothing. Raw text is included in any NBConvert output. NBConvert is a program that will convert your notebook into the other formats such as PDF, see File > Download as > NOTE: using NBConvert requires additional installs: https://nbconvert.readthedocs.io/en/latest/install.html

Python

Python 3.6 Documentation: https://docs.python.org/3.6/ (https://docs.python.org/3.6/)

Python is a language, and like a language has both syntax and nuance. We'll focus on syntax :)

If you're new to programming, hold onto hat as we're going to jump through this QUICKLY!

```
In [ ]:  variable = 'code cell'
         print(f'This is a {variable}')

         variable = 'hello world'
         print(f'{variable}')  # this is a comment and comments don't print
```

Python can also return numbers and do math without needing it to be printed

```
In [ ]:  3
```

```
In [ ]:  3 + 3
```

Math is simple to do with Python, most can be typed in like a calculator, though there is a difference between floating-point and integer division. Floating-point division should be written with floating-point numbers `3.0 / 2.0` and integer vidision with two slashes `3//2` In [ ]:

```
3.0 / 2.0  # floating-point division
```

```
In [ ]:  3 // 2  # integer division
```

```
In [ ]:  3 / 2  # Python 3 defaults to floating-point division
```

Variables can be set singly, or multiple can be defined at one time.

```
In [ ]:  x = 3
```

```
In [ ]:  floating_point_division = x / 2.0
         floating_point_division
```

Defining several variables at once is a process called **unpacking**.

```
In [ ]:  x, y = 5, 3
```

```
In [ ]:  integer_division = x // y
         integer_division
```

Integer division doesn't round up or down, but we can get the remainder with the modulo, which is what is left over after

```
In [ ]:  5 % 3  # modulo
```

Pixels

The pixel is the basic building block of the image and is defined by its intensity (think *value from dark to light*) and possibly its channel. The range of intensities can vary as can the number and type of channels.

Bitonal, or black & white pixels, have two possible intensities: black and white. A grayscale image pixel's intensity varies with the bit-depth of the pixel.

Bit Depth

In an **8-bit grayscale** digital image, the intensity (think *value from dark to light*) of each pixel is defined by 8 individual bits.

A bit can either be on, with a value of 1, or off, with a value of 0.

We can find the total number of possible intensities by finding the total possible mathematical combinations for each bit of the pixel. This is done mathematically by raising the total **bit_possibilities** to the power of the number of **bits_per_pixel**.

```
In [ ]:  x, y = bit_possibilities, bits_per_pixel = 2, 8
         x ** y
```

Exponents can be written using 2 asterisks `x ** y`

```
In [ ]:  possible_intensities = bit_possibilities ** bits_per_pixel
         possible_intensities
```

Functions are bits of code that can be re-used over & over, like Photoshop Actions or Excel Macros. Python has many built-in functions, one of these is `print()`.

You have already seen `print()` used above and we are using what are called *formatted string literals*, which is a scary word for Python will automagically format variables within `{}` brackets when printing. This makes our print statements easier to write and read.

```
In [ ]:  print(f'possible_intensities = {possible_intensities}')
```

```
         range(possible_intensities)  # when iterated over, the range will
         # start with x and end at y - 1 in range(x, y)
```

For loops are an important part of what we will need to iterate over images and can be as simple as counting to 3.

NOTE: Programmers like to start counting with 0!

```
In [ ]:  for number in range(3):  # for x in range(y)
             print(number)  # print x
```

```
In [ ]:  for x in range(possible_intensities):
             print(x)
```

Color Channels

Expanding our discussion to include 24-bit RGB color images, we first break the 24-bits per pixel down into individual color channels. Each 24-bit RGB color image pixel is made up of 3 * 8-bit color channels: Red (R), Green (G), and Blue (B).

To compute the total **possible_intensities** for a 24-bit RGB color image pixel first define the **bits_per_pixel** as **bits_per_channel** * **number_of_channels**.

```
In [ ]:  x, y, z = bit_possibilities, bits_per_channel, number_of_channels = 2, 8, 3
         bits_per_pixel = bits_per_channel * number_of_channels  # 8 * 3

         bits_per_pixel
```

```
In [ ]:  possible_intensities = bit_possibilities ** bits_per_pixel  # 2 ** 24

         possible_intensities
```

So with 3 channels of 256 **possible_intensities** there are 1.67 million possibilities for each pixel in a 24-bit RGB color image.

Moving on . . .

Lists are sequences of items that you can perform repeated actions on the elements of; you can explicitly or programmatically create lists.

```
In [ ]:  list_1 = [3, 4, 1, 0, 44, 13, 9]  # explicitly stated

         list_1
```

Many of Python's built-in functions can be used on lists, such as taking the length of a list, finding the sum of integers in a list, and list manipulation via slicing the list by the index number of the items.

```
In [ ]:  # list_2 is made of up 3 elements:
         # 1. length of list 1
         # 2. sum of a sub-list of list_1 starting at item 0 through item n - 1 (0 - 1)
         # 3. subtract 2 from the last item in list_1 grabbed via index number
         list_2 = [len(list_1), sum(list_1[0:2]), (list_1[-1] - 2)]

         list_2
```

Items can be explicitly or programmatically changed in the list via the index numbers by using square brackets

```
In [ ]:  sublist_1 = list_1[1:3]

         sublist_1
```

```
In [ ]:  sublist_2 = list_1[1:8:3]

         sublist_2
```

```
In [ ]:  print(f'list_1: {list_1}')
         print(f'sublist_1: {sublist_1}')

         sublist_1[1] = 9

         print('******************')
         print(f'list 1 after changing sublist 1: {list 1}')
         print(f list_1 after changing sublist_1: {list_1}')
         print(f'sublist_1 after changing sublist_1: {sublist_1}')

         list_1[1] = 22

         print('******************')
         print(f'list_1 after changing list_1: {list_1}')
         print(f'sublist_1 after changing list_1: {sublist_1}')
```

Tuples are immutable and contain a fixed number of elements.

```
In [ ]:  tuple_1 = (1, 2, 3)

         # tuple_1[1] = 3 will throw a TypeError because tuples cannot be changed
         # Tuples are immutable
         tuple_1[1] = 3
```

Dictionaries contain unordered key-value pairs (if you need an ordered dictionary there is an OrderedDict structure that retains the insertion order, but let's not try to get too far off the path).

```
In [ ]:  dictionary_1 = {'a': 1, 'b': 2, 'c': 3, 'key': 'value'}

         print(f"b: {dictionary_1['b']}")

         'key:', dictionary_1['key']  # calling the key returns the value
```

Sets in Python are like mathematical sets, they contain distinct elements -- all items must be unique!

```
In [ ]:  set_1 = set([1, 2, 3, 4, 3, 2, 1, 1])

         set_1
```

List comprehensions are a quick way of writing for loops when creating a list. Compare them below

```
In [ ]:  print(list_1)

         # create empty list to add to
         squares = []

         for item in list_1:  # don't forget the colon and indention after the for statement!
             square = item * item
             squares.append(square)

         print(squares)
```

```
In [ ]:  print(list_1)

         squares = [item * item for item in list_1]

         print(squares)
```

If statements, like for loops, are a staple of every programming language

You can use an if statement to see if something even exists or not, but be warned it will throw an error if it does NOT exist!

```
In [ ]:  if list_1:
             print(f'list_1: {list_1}')

         if list_17:
             print(list_17)
```

Python will automatically return Boolean values when comparisons are made

```
In [ ]:  len(list_1) > 0
```

```
In [ ]:  for item in list_1:
             if item % 2 == 0 :  # if item is divided by 2 and remainder = 0
                 print(item)
```

```
In [ ]:  odd = [item for item in list_1 if item % 2 == 1]
```

```
odd
```

And the last thing we'll cover in this notebook are functions. Functions have to be defined and can return a result if you so desire.

```
In [ ]:  def is_odd(number):
             """
             Return whether an integer is odd or even
             """
             if number % 2 == 0:
                 return 'even'
             else:
                 return 'odd'

          odd_or_even(13)
```

```
In [ ]: odd_or_even(-8)
```