python_intro

August 2, 2023

1 Python

1.1 Python intro

The w3 school python tutorial starting at the 'syntax' page is a good introduction to the language. Since everyone should be somewhat familiar with at least 1 programming langauge, you should be able to use this as a reference for the basics – writing conditionals, loops, etc.

1.2 Numpy intro

You will also want to familiarize yourself with `numpy`, the package used to extend Python into the world of numeric computing. The numpy beginner's guide is a good place to start, but you should put the user's guide in your favorites.

1.3 Some notes on iteration

It is worth discussing a few things related to loops in a python.

Here is a basic for loop:

[136]:
```
for i in range(5):
    print(f'current iteration: {i}')
```

```
current iteration: 0
current iteration: 1
current iteration: 2
current iteration: 3
current iteration: 4
```

But in python, you will frequently see list comprehensions used in preference to simple for loops. There is no benefit of one over the other necessarily, except for readability and adhering to the python conventions (which aids in readability)

[137]:
```
[print(i) for i in range(5)]
```

```
0
1
2
3
4
```

[137]:
```
[None, None, None, None, None]
```

Note that a list comprehension always returns a list! So, the print statement is executed, but then a list of length 5 with entries all `None` is returned.

The next example is very contrived, but I want to note that sometimes you want to execute some action repeatedly, but don't need to use the index variable (`i` in the examples above). By convention in python, you use `_` instead.

For example:

[138]:
```
for _ in range(5):
    print("Hello, world!")
```

```
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
```

The `range()` operator creates a sequence starting at 0 and ending at 4. This "0-indexed, half-open" behavior needs to be kept in mind in order to be sure that you're extracting the right indicies. For instance, let's say you have the following list:

[139]:
```
# range() returns a generator object, which only gets created in memory
# when it is needed. the list() function forces the generator to create
# the list in memory, where it is stored in the variable my_list.
my_list = list(range(5))

print(my_list)
```

```
[0, 1, 2, 3, 4]
```

and you wish to extract the second, third and fourth items. You would do so like this – note how the list indicies are 1 off what we described above:

[140]:
```
print(my_list[1:4])
```

```
[1, 2, 3]
```

Aside from the comprehension and generator objects like `range()`, Python should seem very familiar, if you are familiar with another language. In general, remember the concepts from functional programming – if you need to do something to each of a set of items, store the items in a list, write a function, and then apply that function over the list. This way, you can write the function using a single object (rather than worry about the entire list) – this makes the development process easier, and the code itself easier to test and maintain.

2 Functions and docstrings

You are going to notice that the functions you will be working on in this class have a good deal of documentation. This is done in one of the common documentation format, sphinx. There are

other common formats – google being one – but we're going to stick to the sphinx format for these assignments.

We are also going to use 'type hints' in the function definition. Python is not a typed language, which means that any variable can take on any datatype at any time. This is great for flexibility, but especially for functions, it is nice to know what datatype the function expects a given input variable to be, and this is what type hints provide. It is just a form of documentation.

For example, this is an entirely valid python function:

[141]:
```
def add_two_numbers(a, b):
        return a + b
```

But, we recognize that documenting our code, especially in a way that can take advantage of automatic documentation builders (like sphinx) is important. Therefore, we add some type hints and write a docstring which provides 'hints' on what the input datatype(s) are expected to be, and what the output datatype(s) are expected to be.

As a side note, by providing type hints, github copilot will be even better at writing your docs for you.

[142]:
```
def add_two_numbers(a: (int,float), b: (int,float)) -> (int,float):
        """Add two numbers together

        :param a: first number
        :type a: int or float
        :param b: second number
        :type b: int or float

        :returns: sum of a and b
        """
        return a + b
```

Something else you'll notice in our functions in the assignments is some checking of types and other features of what we expect the input to be. For example, int his function, we might do this:

[143]:
```
def add_two_numbers(a: (int,float), b: (int,float)) -> (int,float):
        """Add two numbers together

        :param a: first number
        :type a: int or float
        :param b: second number
        :type b: int or float

        :returns: sum of a and b
        """
        if not isinstance(a, (int,float)):
                raise TypeError("a must be an int or float")
        if not isinstance(b, (int,float)):
                raise TypeError("b must be an int or float")
```

```
        return a + b
```

Finally, in packaged code it is very useful to use the python logging package. If you have done any amount of coding in the past, then this is somewhat like using `print` statements. However, it gives you more control over when and where these statements are displayed. In every `assignment.py` file, and every assignment package, a logger is already configured. If you want to use it, you would just add a logger statement, like this:

[144]:
```
# ignore this stuff -- it will be done for you in the assignment
import logging
logger = logging.getLogger('example_logger')
logger.setLevel(logging.DEBUG)
### end ignore

def add_two_numbers(a: (int,float), b: (int,float)) -> (int,float):
        """Add two numbers together

        :param a: first number
        :type a: int or float
        :param b: second number
        :type b: int or float

        :returns: sum of a and b
        """
        # check input
        if not isinstance(a, (int,float)):
                raise TypeError("a must be an int or float")
        if not isinstance(b, (int,float)):
                raise TypeError("b must be an int or float")

        # add a debug logging statement
        logger.debug("Adding %s and %s" % (a,b))

        # return the result
        return a + b

print(f"the output of the function is: {add_two_numbers(1,2)}")
```

```
DEBUG:example_logger:Adding 1 and 2
```

```
the output of the function is: 3
```

Notice that in the example above, we also added inline comments and added some spaces.

Code at this level is for humans, not computers. Smart looking code is code that is readable and easily maintained. It is not code that is difficult to understand and debug.

2.1 Logging

The nice thing about using the logger.debug for printing debug statements is that we can decide not to display those messages by changing the logging level. This way, we can leave useful debug messsages in the code, print them if we wish to, but avoid cluttering the output in general. In the assignments, a logger is set up for you at the top of the `assignments.py` file. So, you can use it in your scripts by just adding lines like:

```
logger.debug("I'm a debug message")
```

or

```
logger.warning("I'm a warning message")
```

If you're using a jupyter console or jupyter notebook to interactively work on the assignment code, then you may want to adjust the logging level. You can adjust the logging level for the assignment packages like this:

```
import logging
import <assignment package name>

<assignment package name>.utils.configure_logging(logging.DEBUG)
```

The levels of logging are:

```
logging.DEBUG
logging.INFO
logging.WARN
logging.ERROR
```

So, continuing with the `add_two_numbers()` example, if we didn't want that debug message to print, we could do one of two things:

1. delete the logger line
2. adjust the logging level

Here is an example of adjusting the logging level

[145]:
```
# NOTE! in your assignment packages, if you are working on them in an
# interactive session, you will use code similar to the snippet above to
# adjust the logger level. This is only for demonstration purposes in this
# notebook.
logger.setLevel(logging.INFO)

print(f"the output of the function is: {add_two_numbers(1,2)}")
```

```
the output of the function is: 3
```