MR2020: Coding for METOC

Module 3: Miscellaneous Items

Table of Contents

Operators

Python contains many different types of operators:

- Arithmetic (`+  -  *  /  **  %  //`)
- Comparison (`>  <  <=  >=  ==  !=`)
- Assignment (`= += -= *= /= **= %= //=`)
- Logical (`and  or  not`)
- Identity (`is  is not`)
- Membership (`in  not in`)
- Bitwise (`&  |  !  ~  <<  >>`)#

Bitwise operators not covered in detail in this class.

Arithmetic

```
+   Addition: A + B == 5.

-   Subtraction: A - B == 1.

*   Multiplication: A * B == 6.

/   Division: A / B == 1.5.

**  Exponent: A ** B == 9.

%   Modulus:  A % B == 1

//  Floor division: A // B == 1
```

Assume

```
A = 3
B = 2
```

NOTE: None of the arithmetic operators change A or B. They just do a calculation with them.

Modulus calculates the remainder when dividing.

Floor division rounds down to nearest integer.

Comparison

```
>  Greater than: A > B == True.

<  Less than: A < B == False.

<= Less than or equal to: A <= B == False.

>= Greater than or equal to: A >= B == True.

== Equal to: A == B == False.

!= Not equal to: A != B == True.
```

Assume

```
A = 3
B = 2
```

Assignment

```
=  Is assigned: A = 3.

+= Add and reassign: A += B makes A == 5.

-= Subtract and reassign: A -= B makes A == 1.

*= Multiply and reassign: A *= B makes A == 6.

/= Divide and reassign: A /= B makes A == 1.5.

**=Exponent and reassign: A **= B makes A == 9.

%= Take modulus and reassign: A %= B makes A == 1.

//= Floor divide and reassign: A //= B makes A == 1.
```

Assume

```
A = 3
B = 2
```

NOTE: Assignment variables define or change the value of a variable on the left-hand side of the operator.

Logical

`and` Returns `True` if both statements are `True`.

```
This is False:  A > 2 and B < 1
```

`or` Returns `True` if either statement is `True`.

```
This is True:   A > 2 or B < 1
```

`Not` Reverses result and returns `True` if the statement is `False`.

```
This is False:  not A > 2
```

Assume

```
A = 3
B = 2
```

What happens if we do this? `A > 2 or B < 1 and not B < 0`

Identity

```
# Example variables
a = [1, 2, 3]
b = a
c = [1, 2, 3]

# is operator
if a is b:
    print("a and b point to the same object")

# is not operator
if a is not c:
    print("a and c do not point to the same object")
```

`is` and `is not` are used to determine if two variables point to the same object. `b` points to the same object as `a` because it is assigned as `a`. Even though `c` is set to an equivalent list as `a`, it occupies a different space in memory, so `c is not a`.

Try running `a.append(4)` and see what happens to `b`.

Membership

```
# Example list
fruits = ["apple", "banana", "cherry"]

# in operator
if "banana" in fruits:
    print("Banana is in the list of fruits")

# not in operator
if "grape" not in fruits:
    print("Grape is not in the list of fruits")
```

`in` and `not in` are useful for identifying if something is an element in a list, set, or even NumPy array. The example above evaluates whether a string is or is not in a list, but this could be extended to determining if a number is in an array.

Importing Modules and Libraries/Packages

A **module** is a single portable piece of reusable code (a .py file) that contains functions, classes, and variables.

A **package** is a collection of related modules stored in a common directory tree. This term is often used interchangeably with **library.** We can import entire packages, but more commonly, we import specific modules from a package.

****Modules take time to load. Whenever possible, only import necessary modules! This will also make your code more compact and readable.****

TIP: Combine all of your module imports at the top of your code.

How to Import

Simple import of a module:

```
import datetime
```

Import a class, function, or attributes from a module or package

```
from datetime import datetime
from numpy import array
from math import sqrt, pi
```

Import a module or package with an alias (for example I can later type `np.` instead of `numpy` whenever I want to call a method from NumPy in my code.

```
import numpy as np
import pandas as pd
```

Combine `from` , `import`, and `as`. The following two lines are equivalent.

```
from matplotlib import pyplot as plt
import matplotlib.pyplot as plt
```

Creating a Module

You can also create your own module. This may be useful if you have a collection of classes and/or functions that you have created yourself and want to use in various codes but don't want to copy/paste into each code individually. For example, you could create a file called thermodynamics.py in the same directory where your code is running, and the contents might look like this:

```
def thetatoT(theta,p):

# theta in K, p in hPa.

p00, R, cp = 1000, 287, 1004
return theta / ( (p00/p)**(R/cp) )


#**************************************

def Ttotheta(T,p):

# T in K, p in hPa.

p00, R, cp = 1000, 287, 1004
return T * ( (p00/p)**(R/cp) )
```

If we did this in a different code:

```
import thermodynamics as thm

thm.thetatoT(313,925)
```

We would get 306.1 without having to copy and paste the functions inside of thermodynamics.py into the new code.

Installing Packages into Existing Python Environment

Sometimes, we may need to use packages that we did not originally install into our mr2020 environment during setup on the first day of class. In this case, we would need to enter the terminal and install the packages. There are two approaches to this:

1) Install the package to the existing environment created in Module 0 (mr2020)
   Pros: All packages conveniently located in a single environment
   Cons: Packages might have conflicts with each other, effectively breaking the environment.

2) Install an entirely new environment
   Pros: Prevent package incompatibility (e.g., one package calls a depreciated method from another package)
   Cons: Can end up with several environments with difficulty remembering which environment should be used for each task

Creating a New Environment

The instructions for installing a new environment are found in Module 0. To review, the command to create the mr2020 environment was (one line):

conda create –c conda-forge –n mr2020 numpy scipy pandas matplotlib ipython metpy xarray ipykernel

-c indicated the package repository to use and -n denoted the name of the new environment on your local machine. Everything after mr2020 was a list of the packages to be installed. You could add to this list, but you would not be able to use the name mr2020 because you have already created an environment with this name. Conda then automatically installs several other packages that the ones listed depended upon.

Add to Existing Environment

Ensure that the mr2020 environment is activated in Visual Studio Code, then access a terminal. At the start of the line that pops up in the command terminal, you should see (mr2020) including the parentheses. If not, enter

```
conda activate mr2020
```

Then, once you are sure you are in the mr2020 environment, do the following

```
conda install -c conda-forge PKGNAME
```

Replacing PKGNAME with the actual name of the package. Sometimes, the package is not found in the conda-forge repository, and you may need to use pip. If this is the case, the package can be installed using the following command (you may need to first install pip using conda like in the above command):

```
pip install PKGNAME
```