Code Flow

Carver J. Bierson

June 26, 2017

The power of computer comes from their ability to perform many many calculations and make decisions about what to do next. The main method for doing these repeat calculations is loops with logic statements

1 Conditionals

The other key programming tool is conditional statements. This is usually done via *if-else* statements. An *if* statement works by checking some logical statement. If that statement is true, it does the indented section of code. i.e.

```
a=4 # Define a
if a<0:  # Is a less than 0?
    print('a is negative') # if so write this
elif a==0: # Does a equal 0?
    print('a is 0') # If so do this
else: # this happens if none of the previous statements are true
    print('a is positive')
```

You can nest *if* statements within one another for more complex conditions

```
a=4 # Define a
a=40 # Define b
if a<0:  # Is a less than 0?
    if b<0:  # Is b less than 0?
        print('a and b are negative') # if so write this
    elif b==0: # Does b equal 0?
        print('a is negative, b is 0') # If so do this
    else: # this happens if none of the previous statements are
    true
        print('a is negative, b is positive')
```

or you can do the same thing with more complex logic checks

```
a=4 # Define a
a=40 # Define b
if a<0 and b<0:  # are a and b both negative
    print('a and b are negative') # if so write this
elif a<0 and b==0: # Does a equal 0?
    print('a is negative, b is 0') # If so do this
elif a<0 and b>0: # this happens if none of the previous statements
 are true
    print('a is negative, b is positive')
```

Which of these two methods you use is up to you. I recommend that you do whichever you think will make your code easier to read. Expressions are evaluated in order of arithmetic (add, multiply, etc.), comparisons (less than, greater than), logic (and, or).

Table 1: Common logic syntax

check if	syntax
Equal to	==
Less than	<
Greater than	>
Less than or equal to	<=
and	*and*
or	*or*
not	*not*
a is member of *b*	*in*

2 Loops

Loops allow you to repeat a task a set number of times (for loops) or until some condition is met (while loops).

In python the basic for loop syntax is

```python
# This loop prints out the numbers 0-4
for i in range(5):
    print('{:d}'.format(i))
```

The way to think of this loop is as follows. *range* creates a list of 5 values, [0,1,2,3,4]. *i* then takes on each of these values in order. The indented section of code happens each time *i* takes on a new value. In this case it prints out the value of *i*. After the loop finishes *i* will still have a value of 4.

> **Question**: Is the following loop valid? If so what would it do?
>
> ```python
> for i in 'Hello':
>     print(i)
> ```

We can do the same thing using a while loop.

```python
# This loop prints out the numbers 0-4
i=0
while i<5:
    print('{:d}'.format(i))
    i+=1
```

Here we first have to give *i* a starting value. Each time through the loop *i* is printed than we increment it by 1 (i+=1 is the same as i=i+1). Note if you are not careful with while loops they may never break.

> **Question**: Rewrite the above while loop so that it prints the same values but finishes with *i*=4.
>
> **Question**: Rewrite the hello loop above as a while loop.

3 Control statements

Sometimes you want to skip a value of a loop or leave the party early. For this we have some syntax that is sometimes called control statements. *break* leaves a loop entirely.

```
# This loop prints out numbers
for i in range(5):
    if i>2:
        break
    print('{:d}'.format(i))
```

will print

```
0
1
2
```

continue skips just that iteration of the loop

```
# This loop prints out numbers
for i in range(5):
    if i==3:
        continue
    print('{:d}'.format(i))
```

will print

```
0
1
2
4
```

pass is usually used as placeholder when you are structuring code but haven't yet fleshed it out. Nothing happens when you call pass

```
# This loop is invalid and will cause an error
for i in range(5):

# This loop is valid, but doesn't do anything
for i in range(5):
    pass
```

4 Logical indexing

It is much faster in python to not loop over all variable in an array but instead apply operations to via logical indexing. Logical indexing is using an array of Boolean (True/False) values to select a subset of elements from an array (instead of a *start:end* syntax).

```
>>> a=np.linspace(1,10,10)
>>> a
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
>>> a>5
array([False, False, False, False, False,  True,  True,  True,
    True,  True], dtype=bool)
>>> a[a>5]
array([  6.,   7.,   8.,   9.,  10.])
>>> b=np.linspace(10,1,10)
>>> b
array([ 10.,   9.,   8.,   7.,   6.,   5.,   4.,   3.,   2.,   1.])
>>> np.logical_and(a>5,b>4)
array([False, False, False, False, False,  True, False, False,
    False, False], dtype=bool)
>>> a[np.logical_and(a>5,b>4)]
array([ 6.])
>>> a[np.logical_and(a>5,b>4)]=-20
>>> a
array([  1.,   2.,   3.,   4.,   5., -20.,   7.,   8.,   9.,  10.])
```

5 Functions

When writing code you often have tasks you want to repeat in different parts of the code. By writing functions you condense these repeat sections into one place making debugging easier. Your code will also be more readable if you move different sections into well named functions. Functions are also great because if you write them well you can take a function from one project and use it in another (even within this class).

```
 # Define an addition function
def add(a,b):
  return (a+b)
# use our new function
print(add(7,2))
```

will print out 9. As written this function takes in two variables and returns their sum. The user must always pass two, and only two, variables for this to work. If I try to just type *add(7)* I get

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: add() takes exactly 2 arguments (1 given)
```

We can write functions where some values are optional, i.e.

```
 # Define an addition function
def add(a,b=1):
  # This function adds the two values a and b.
  # If only one value is given it increments the value by 1.
  return (a+b)
# use our new function
print(add(7))
```

will return 8. If we pass to variables, like *add(7,2)* we overwrite the default value and still get 9.

5.1 Returning multiple values

It is not uncommon to want to return multiple values from a function. To understand how to do this in python it is import to talk about *tuples*. *tuples* are similar to lists in that they can contain multiple elements of different types. Tuples are defined by parentheses instead of brackets.

```
tuple1=(5,'hello') # This creates a new tuple

print(tuple1[1]) # This should print hello
```

Tuples are useful because they can be *unpacked* into multiple variables.

```
 value1, string1=tuple1 # unpack tuple

print(string1) # This should print hello
```

This could be used for a coordinate transform function

```
 import numpy as np
 # Define a function to go from cartesian to polar coordinates
def cart2polar(x,y):
  r=np.sqrt(x**2.0+y**2.0)
  theta=np.arctan(y/x)
  return (r,theta)
```

```

# use our new function
r,theta=cart2polar(1.0,1.0)

# Print out the results
print(r)
print(theta)
```

Note that you could capture all the returned values in their tuple form.

5.2 Variable scope

A function only knows about variables passed to it or defined within the function. This does mean that a variable within a function can have the same name as variables outside the function and those can have different values at the same time. An example is

```
 # Define an addition function
def SimplePrint5():
  # This function defines a variable a and prints it
  # This function takes no arguments
  a=5
  print(5)
# use our new function
a=10
SimplePrint5()
```

will still print out 5 because the a in the function is separate from the value of a outside the function. This said if you give two things the same name in a different workspace have a good reason for doing so. **The readability of your code is important!**.

5.3 Modules

In this class we have already used the built in functions from the numpy module. It is often helpful to write your own modules with functions that you use in multiple scripts. If you write set of functions in the file *MyModule.py*, you can load them into another script using *import MyModule as mm*. Note the module file must be in the same directory as your script file.

6 Classes

In python almost everything is an object. Objects have attributes (data values) and methods (built in functions that act on the object). For this class we will not be using custom classes very much but it is good to have a basic idea how they work.

```
 import numpy as np
 # Define the class "Circle"
class Circle:
    # This happens when a "Circle" object is created
    def __init__(self, x0=0.0, y0=0.0, r=1.0):
        # Note that optional values are defined the same as regular
     functions
        self.x0=x0 # Circle x center
        self.y0=y0 # Circle y center
        self.r=r # Circle radius

    def area(self): # this is a method that takes no inputs
        # The area method prints out the area of the circle
        print(2.0*np.pi*self.r**2)

    # This method returns the xy location of point on the circle at the input radian
    def rad2cord(self, radian):
        x=self.x0+self.r*np.cos(radian)
        y=self.y0+self.r*np.sin(radian)
        return (x,y)


# Now I will use the circle class
Rad5Circle=Circle(r=5) #x0 and y0 will be their default values

print('x0={:f}, y0={:f}, r={:f}'.format(Rad5Circle.x0, Rad5Circle.y0, Rad5Circle.r))

Rad5Circle.area() # print out area, Note self is automatically passed to methods

x1,y1=Rad5Circle.rad2cord(np.pi/2.0) # x1 should=0, y1 should =5

print('x1={:f}, y1={:f}'.format(x1,y1))
```

7 Concluding thoughts

You now have all of the basic programming tools we will use in this class. Numerical modeling is just the art of combining arrays, loops, and conditionals in ways to solve problems. Next we will learn how to make these methods interact with outside data (reading/writing data and making plots). After that this course will be learning to apply these tools to different problems.

8 In Class Practice

The Fibonacci number of a value is the sum of the integers preceding it,

$$F_n = F_{n-1} + F_{n-2} \tag{1}$$
$$F_0 = 1, F_1 = 1 \tag{2}$$

- Write a loop that calculates Fibonacci numbers 1-10 and prints them.
- Write a function using your loop that calculates Fibonacci number n.
- Write a function that calculates Fibonacci number n recursively.