Flow control in Python

Previously, all you saw was statements in a single file which were executed only in the order they appear. This will suffice for simple problems, but often you may want to react to some value unknown at the time of writing, or run some similar code many times or with slight variations.

If statements

```
if condition:
    ...
```

Whatever code appears in the `...` will only be executed if `condition` evaluates to `True`. For example:

```
some_values = [4, 5, 'test']
if 'test' in some_values:
    print('found it!')
```

or

```
if 5*6 < 30:
    print('I broke math')
```

The `if` is the only required part of the full syntax, which includes an optional `else`:

```
if 'test' in some_values:
    print('found it')
else:
    print('noooooooo')
```

There can also be any number of `elif` statements between an `if` and an `else`, and there need not be an `else` after the last `elif`.

```
test_value = 40
if test_value < 10:
    print('huh')
elif test_value < 20:
    print('interesting')
elif test_value < 30:
    print('something new')
else:
    print('oh well')

print('this always runs')
```

The `if` and `elif` conditions are checked in the order they appear. If any of them evaluate to true, the code in the first true statement's block is run, and program executon skips to the end.

For loops

```
for item in iterable:
    ...
```

The for loop in Python requires two things: a loop variable and something that is iterable. When the loop starts, the first value of `iterable` is assigned to `item` and the code in the block executes. At the end of the block, the next value in `iterable` is assigned to `item`, and the block runs again, until the end of `iterable` is reached. For example:

```
for filename in os.listdir('.'):
    print(filename)
print(filename)
```

Notice that the last name in the current directory is printed twice here. The loop variable retains the value of the last item in the iterable when the loop exits.

If you are familiar with other programming languages, this loop syntax may seem a bit unusual. This is because Python doesn't have a traditional for loop, instead it only has for each loops. This can be a bit unintuitive for those with other programming backgrounds, and people often fall into traps like:

```
for i in range(len(some_iterable)):
    print(some_iterable[i])
```

Where in Python we can just do

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

Now which is easier to read? Which do you think is faster? The second is actually *way* faster. Instead of just pulling the things we want from an iterable, we generate another iterable of indices, pull an index from from the new `range` , and then index the thing we want to pull elements from. Now while this is faster, the real benefit to us is the readability. It's very uncommon for the speed of any data processing program to be bound by Python itself. If your code is bound by the speed of the for loop itself and not what's going on inside, you're almost certainly doing something wrong.

If you need to iterate across multiple iterables at once, you can do that using tuple unpacking and `zip` (or `izip`):

```
for name, date in zip(some_names, some_dates):
    print(name, date)
```

Packing/Unpacking

In the example above, I use `zip` to create another iterable of tuples, then unpack the tuples into `name` and `date` . Tuple unpacking can also let you do things like swap the values of two variables:

```
x = 1
y = 2
x,y = y,x
```

In general this works will all ordered data types, but is mostly used with tuples.

If you really insist though that you need the indices of of the iterable you are iterating on, Python offers `enumerate` .

```
for f, filename in os.listdir('.'):
    print(f, filename)
```

While loops

The while loop combines the loop idea with some boolean (true/false) test.

```
number = 1
while number < 10:
    number += 1
    print(number)
```

The danger with while loops is that you can put yourself in a situation where you write something like:

```
while test < 10:
    ...
    test = 5
    ...
    test += 1
```

This loop will never terminate. That's not inherently a bad thing, since Python provides two other keywords for working with loops: `break` and `continue`.

Break and Continue

If you want to write a loop that runs at least once, you can do this with a while loop and `break`.

```
while True:
    ...
    if test_condition:
        break
```

The break keyword immediately exits the current loop. For example, this is still an infinite loop:

```
while True:
    while True:
        break
```

`continue` skips the rest of that loop iteration. For example:

```
for name in os.listdir('.')
    if name.endswith('.py'):
        continue
    print(name)
```

There is a much better way to accomplish what I just demonstrated with `continue`, but `continue` has very limited uses.

Aside: Indentation

Python *requres* indentation in a way most languges do not (except for scripting languages). This is known as significant whitespace, and can be a problem in some situations. For example, you can indent with tabs or spaces, but cannot mix them together within a single file. Most text editors can be configured to insert spaces when the tab key is pressed. Spyder does this by default. Python doesn't specify how much to indent, only that you need to. Indention of 4 space is typical in Python, though some languages prefer 2 or 8. The usual argument is that 2 is not clear enough, and with 8 spaces you can spend a significant fraction of the screen's width on indentation.

Functions

Functions are one of the most powerful tools you'll encounter in programming, particularly for data processing. You've actually been using functions already; any name followed directly by parentheses is a function, and a statement where I put something in the parentheses is a function call. To define your own function, do the following:

```
def function_name(argument1, argument2):
    test = argument1 - argument2
    return test**2
```

Now I have defined a function called `function_name`, which takes two arguments. If I have two variables `spam` and `eggs`, I can now say `something = function_name(spam, eggs)`.

One of the functions you've been using is `astropy.io.fits.open` , which we've imported as just `fits.open` . This function takes a file name (or path to a file) and opens it, and returns an `HDUList` .

This might sound confusing but except for the syntax to define functions I promise you've seen this before. The syntax for calling functions was chosen for a reason, it mimics the notation used in math. This does not however mean you should even call a function `f` . In Python the burden is on the person writing the code to use names for things that make it clear what they are. `f` if completely meaning-free. Even if nobody else is going to see your code future you will probably be annoyed at past you for doing something so unhelpful.

Just as in math, a function is a variables all on its own that can be manipulated. In technical terms, we'd say that Python's functions are first-class functions (or first-class citizens). A function is just a normal variable, except it has this extra property where if you put parentheses after it you invoke some other behavior. So you can have functions that take functions as arguments and return functions. This sounds wacky but it's something that turns out to be very useful, though we'll not talk about it for now because it isn't imminently useful for data processing.

Take a look at the documentation for `fits.open` : http://docs.astropy.org/en/stable/io/fits/api/files.html#open Right there on the second line you see the function and all its arguments laid out. Notice that while we've just been passing one argument, there are a bunch of others here, but all with `name=a_thing` and there's also a `**kwargs` . First to the `=` arguments. Those are keyword arguments, which are optional and therefore require a default value when the function is defined. The defaults are what are shown. The arguments without a `=` are known as positional arguments; position matters. If I were to call `function_name(spam, eggs)` I may not get the same thing back as `function_name(eggs, spam)` . On the other hand, if I were to call `fits.open('test_image.fits', memmap=True, cache=False), that will do the same thing as` fits.open('test_image.fits', cache=False, memmap=True) `. Keyword arguments are super nice because they increase the readability of your code. Remember in the introduction I did something like` np.clip(image, 0, 50) `; it's totally unclear what the` 0 `and` 50 `are, and only slightly apparent what` image `is. If you check the documentation for` numpy.clip `you'll see that there is no` = `next to the arguments for the max and min, but that doesn't stop me from using them. I can do` np.clip(image, a_min=0, a_max=50) `. Now isn't that much more clear? If you happen to disagree with the ordering of the arguments, using their names lets you change that around to suit you:` np.clip(image, a_max=50, a_min=0)` will return the same array.

When and where to use functions

You should use functions as often as possible. Ideally, a project of any size will be a short bit of procedural code and a bunch of short function definitions. This helps keep your code modular. Good modular code is broken up into many small pieces with good names so you can navigate it quickly and avoid writing the same line or even the same idea in multiple places. That's the whole point of flow control; the programmer should only have to program one idea once, then using flow control tools you can apply it in many circumstances. Good use of functions will speed up your development time. It may seem like extra work to put things into functions instead of copying and pasting code but future you will be very annoyed with past you if current you is lazy. By "use functions as often as possible" I really mean it. At any point in the future you may want to use some code you wrote before. If it's packaged up into a function it's almost effortless to use it in a new project.