The cell above loads the visual style of the notebook when run.

In [1]:
```
from IPython.core.display import HTML
css_file = '../styles.css'
HTML(open(css_file, "r").read())
```

Out[1]:

Making Choices

Learning Objectives

- Explain the similarities and differences between tuples and lists.

- Write conditional statements including `if`, `elif`, and `else` branches.
- Correctly evaluate expressions containing `and` and `or`.

In our last lesson we saw how we could find datasets containing variable stars by drawing plots. How can we use Python to recognise the features we saw, and take a different action when it finds them? In this lesson, we'll learn how to write code that runs only when certain conditions are true.

Conditionals

We can ask Python to take different actions, depending on a condition, with an if statement:

In [2]:
```
num = 37
if num > 100:
    print('greater')
else:
    print('not greater')
print('done')
```

```
not greater
done
```

The second line of this code uses the keyword `if` to tell Python that we want to make a choice. If the test that follows the `if` statement is true, the body of the `if` (i.e., the lines indented underneath it) are executed. If the test is false, the body of the `else` is executed instead. Only one or the other is ever executed:

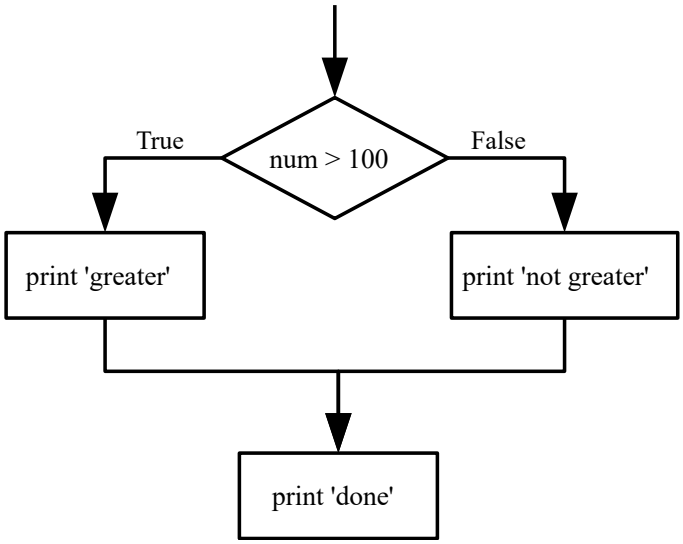

Figure: Executing a conditional

Conditional statements don't have to include an `else`. If there isn't one, Python simply does nothing if the test is false:

In [3]:
```
num = 53
print('before conditional...')
if num > 100:
    print('53 is greater than 100')
print('...after conditional')
```

```
before conditional...
...after conditional
```

We can also chain several tests together using `elif`, which is short for "else if". The following Python code uses `elif` to print the sign of a number.

In [4]:
```
num = -3
if num > 0:
    print(num,'is positive')
elif num == 0:
    print(num,'is zero')
else:
    print(num,'is negative')
```

```
-3 is negative
```

One important thing to notice in the code above is that we use a double equals sign `==` to test for equality rather than a single equals sign because the latter is used to mean assignment.

We can also combine tests using `and` and `or`. `and` is only true if both tests are true:

In [5]:
```
if (1 > 0) and (-1 > 0):
    print('both tests are true')
else:
    print('one test is not true')
```

```
one test is not true
```

while `or` is true if at least one test is true:

In [6]:
```
if (1 > 0) or (-1 > 0):
    print('at least one test is true')
```

```
at least one test is true
```

Checking our Data

Now we've seen how conditionals work, we can use them to look for variable stars in our star data.

When there was no variable star present in our dataset, the "processed data" for each showed nothing but random noise, and the scatter of each star looked to be similar. The exception was the fourth dataset, where one star seemed to vary more than the others.

How might we test for this? As a reminder,

```
processed_data.std(axis=1)
```

will calculate a numpy array of standard deviations, one for each star. If a star is variable, it will have a higher standard deviation than the other stars.

In [7]:
```
import numpy

#load data
data = numpy.loadtxt(fname='data/star_data_04.csv', delimiter=',')

# calculate the average brightness at each time, over all stars (rows)
ave_brightness = data.mean(axis=0)

# divide by the average brightness
processed_data = data/ave_brightness

# find the standard deviation of each star
deviations = processed_data.std(axis=1)
print(deviations)
```

```
[ 0.07354632  0.0783064   0.06865498  0.07188308  0.07207184  0.07693856
  0.07701755  0.07000342  0.07514565  0.07822513  1.22748527  0.07364551
  0.07889032  0.07324603  0.07469193  0.07667576  0.07701152  0.06589979
  0.06935011  0.06868695]
```

We can see that one star has a much higher standard deviation than the others. How might we use Python to find such an outlier?

Once we have an array of standard deviations for each star, we can look for ones much greater than the mean:

In [8]:
```
mean_deviation = deviations.mean()
std_deviations = deviations.std()
for star_deviation in deviations:
    if (star_deviation - mean_deviation > 3.0*std_deviations):
        print('variable star in this data')
```

```
variable star in this data
```

Putting this together with the `for` loop from the last session we have:

In [9]:
```
import glob

filenames = glob.glob('data/*.csv')
for f in filenames:
    print (f)

    data = numpy.loadtxt(fname=f, delimiter=',') # load in the data

    # calculate the average brightness over all stars (rows)
    ave_brightness = data.mean(axis=0)

    # divide by the average brightness
    processed_data = data/ave_brightness

    # standard deviation of each star
    deviations = processed_data.std(axis=1)

    mean_deviation = deviations.mean()
    std_deviations = deviations.std()
    for star_deviation in deviations:
        if (star_deviation - mean_deviation > 3.0*std_deviations):
            print('variable star in this data')
```

```
data/star_data_01.csv
data/star_data_02.csv
data/star_data_03.csv
data/star_data_04.csv
variable star in this data
data/star_data_05.csv
data/star_data_06.csv
data/star_data_07.csv
data/star_data_08.csv
data/star_data_09.csv
data/star_data_10.csv
data/star_data_11.csv
data/star_data_12.csv
```

How many paths?

Which of the following would be printed if you were to run this code? Why did you pick this answer?

1. A
2. B
3. C
4. B and C

```
if 4 > 5:
    print 'A'
elif 4 == 5:
    print 'B'
elif 4 < 5:
    print 'C'
```

Write your answer here

4 is less than five so 'C' will be printed

In [2]:
```
if 4 > 5:
    print('A')
elif 4 == 5:
    print('B')
elif 4 < 5:
    print('C')
```

```
C
```

What is truth?

`True` and `False` are special words in Python called `booleans` which represent true and false statements. However, they aren't the only values in Python that are true and false. In fact, *any* value can be used in an `if` or `elif`. After reading and running the code below, explain what the rule is for which values are considered true and which are considered false. (Note that if the body of a conditional is a single statement, we can write it on the same line as the `if`.)

In [3]:
```
if '': print ('empty string is true')
if 'word': print ('word is true')
if []: print ('empty list is true')
if [1, 2, 3]: print ('non-empty list is true')
if 0: print ('zero is true')
if 1: print ('one is true')
```

```
word is true
non-empty list is true
one is true
```

> ***Write your answer here***

1) ' ' is given. This is not an empty string because there is a space key character inside it

2) The string has 'word' in it and returns "word is true" so the condition must be that the string is 'word'

3) empty list is true is not printed so the list [] is not empty

4) The list is non empty so the condition prints out "non-empty list is true"

5) if the value is 0 "zero is true" is printed, this isn't true so it isn't printed

6) the value is one and so one is true

✎Close enough

> Write some conditions that print `True` if the variable `a` is within 10% of the variable `b` and `False` otherwise. Compare your implementation with your partner's: do you get the same answer for all possible pairs of numbers?

In [13]:
```
# Write your code here

a = 9
b = 8
#c = .1*b

if (b < 1.1*a) and (b > 0.9*a):
    print('true')
elif b == a:
    print('false')
elif :
    print('false')

#if (c <= a) or ( a <= c):
    #if c <= a <= c:
#    print('true')
#elif (c < a) and (a < c):
#   print('false')
```

```
  File "<ipython-input-13-0e7f4027d7bd>", line 11
    elif:
       ^
SyntaxError: invalid syntax
```

Miscellany

In place operators

Python (and most other languages in the C family) provides in-place operators (reference.html#in-place-operator) that work like this:

```
x = 1  # original value
x += 1 # add one to x, assigning result back to x
x *= 3 # multiply x by 3
print x
```

```
6
```

Write some code that sums the positive and negative numbers in a list separately, using in-place operators. Do you think the result is more or less readable than writing the same without in-place operators?

In [28]:
```
# WRITE YOUR CODE HERE

# varibles
x = 1
y = -2
z = 3
i = -4

pos = 0
neg = 0

list = [x,y,z,i]

for i in list:
    if i < 0:
        neg += i
    elif i > 0:
        pos += i
    elif i == 0:
        i = i

print(neg, pos)
```

```
-6 4
```

Exchanges

What is the overall effect of the code below?

```
left = 'L'
right = 'R'

temp = left
left = right
right = temp
```

Compare it to

```
left, right = right, left
```

Do they do the same thing? Which do you find easier to read?

Write your answer here

Temp is set = to 'L', left is now 'R', and right becomes 'L'

The code below sets left = right and right = left.

The bottom code is simpier and easier to read.

In [21]:
```
left = 'L'
right = 'R'

#temp = left
#left = right
#right = temp

left, right = right, left

print(left, right)
```

```
R L
```

In [ ]:
```

```