Python Programming for Data Science

Week 37, Monday

- String formatting
- Conditionals
- Loops

Recap: datatypes — a popquiz

- How do you find the length of a list?
- How do you extract a sub-list from a list?
- What is the "opposite" of the `split` string method?
- How do I create an empty dictionary?
- How do I extract the 2nd element from a dictionary?

Recap: datatypes — a popquiz

- How do you find the length of a list?
- How do you extract a sub-list from a list?
- What is the "opposite" of the `split` string method?
- How do I create an empty dictionary?
- How do I extract the 2nd element from a dictionary?
  Trick question!

A short aside: The Python console

Python has an interactive console

You can start it by typing `python` from the bash prompt.

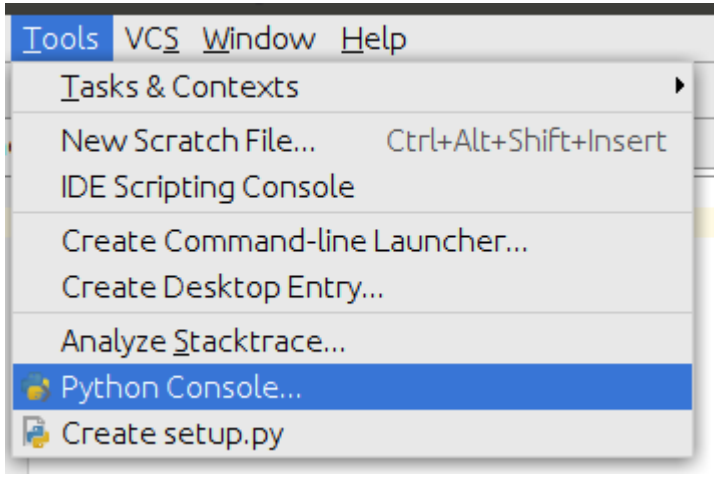
```
$ python
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python has an interactive console

You can start it by typing `python` from the bash prompt.

```
$ python
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

...or from within Pycharm:



Python console: evaluation

In the console, Python will print out the result of any expression you enter

Python console: evaluation

In the console, Python will print out the result of any expression you enter

Enter a number:

```
>>> 10
10
```

Python returns the same number back.

Python console: evaluation (2)

Enter a sum:

```
>>> 10+20
30
```

Python calculates the sum.

Python console: evaluation (3)

Or any other Python expression:

```
>>> len([1,2,3,4])
4
>>> "1 2 3 4".split()
['1', '2', '3', '4']
```

Python console: what's it good for?

The console is a nice way to very quickly test single Python expressions.

Python console: what's it good for?

The console is a nice way to very quickly test single Python expressions.

...but as soon as you are writing more than a few lines, it will typically be easier to use Pycharm.

Python console: what's it good for?

The console is a nice way to very quickly test single Python expressions.

...but as soon as you are writing more than a few lines, it will typically be easier to use Pycharm.

...and we expect all handins in this course to be handed in as `.py` files (not as transcripts from an interactive session).

Python console: printing?

One more thing that sometimes leads to confusion:

Q: Why don't we need to use `print`?

```
>>> len([1,2,3,4])                    # why does this work?
4
```

Python console: printing?

One more thing that sometimes leads to confusion:

Q: Why don't we need to use `print`?

```
>>> len([1,2,3,4])                    # why does this work?
4
```

A: The console will evaluate **and automatically print** the result of any expression.

Python console: printing?

One more thing that sometimes leads to confusion:

Q: Why don't we need to use `print`?

```
>>> len([1,2,3,4])                    # why does this work?
4
```

A: The console will evaluate **and automatically print** the result of any expression.

Note that this is different from the behavior in a Python script.

String formatting

Types - Strings - formatting

You can insert values at certain places in a string using *string formatting*.

Two ways of string formatting in Python:

- old way - operator %
- new way - str.format()

String formatting - % operator

```
s = "Name: %s, Height: %f" % ("Barack Obama", 1.85)
print(s)
```

```
'Name: Barack Obama, Height: 1.85'
```
output

`%s` and `%f` are called *conversion specifiers*, and tell Python how the values should be interpretted

String formatting - % operator

```
s = "Name: %s, Height: %f" % ("Barack Obama", 1.85)
print(s)
```

```
'Name: Barack Obama, Height: 1.85'
```
output

%s and %f are called *conversion specifiers*, and tell Python how the values should be interpretted

Most commonly used conversion specifiers:

%s	string
%d	integer
%f	float

String formatting - % operator (2)

In most cases, you can just use the conversion specifier %s, which will use the default string representation of the value that you are inserting

```
print("Name: %s, Height: %s" % ("Barack Obama", 1.85))
```

```
'Name: Barack Obama, Height: 1.85'
```
output

String formatting - % operator (2)

In most cases, you can just use the conversion specifier %s, which will use the default string representation of the value that you are inserting

```
print("Name: %s, Height: %s" % ("Barack Obama", 1.85))
```

```
'Name: Barack Obama, Height: 1.85'
```
output

...but for detailed control of the output, use the appropriate conversion specifier:

```
# print height with 4 digits after comma
print("Name: %s, Height: %0.4f" % ("Barack Obama", 1.85))
```

```
'Name: Barack Obama, Height: 1.8500'
```
output

More on conversion specifiers: the Python documentation.

String formatting - % operator - by name

You can also refer to the values by name

```
print("I have %(n_apples)s apples and %(n_pears)s pears." % {"n_apples":5,
                                                            "n_pears":7})
```

```
'I have 5 apples and 7 pears.'
```
output

Note the use of a dictionary on the right.

String formatting - % operator - by name

You can also refer to the values by name

```
print("I have %(n_apples)s apples and %(n_pears)s pears." % {"n_apples":5,
                                                            "n_pears":7})
```

```
'I have 5 apples and 7 pears.'
```

output

Note the use of a dictionary on the right.

This can be convenient if you want to insert the same value many times:

```
print("3%(sep)s14 2%(sep)s72 1%(sep)s41" % {"sep":"."})
```

```
'3.14 2.72 1.41'
```

output

String formatting - .format()

Python 3 introduced a new string formatting technique:

```
"Name: {}, Height: {}"
          .format("Barack Obama", 1.85)
```

output

```
'Name: Barack Obama, Height: 1.85'
```

Within the curly brackets, you can:

Specify a positional index:

```
"Name: {1}, Height: {0}".format(
    "Barack Obama", 1.85)
```

output

```
'Name: 1.85, Height: Barack Obama'
```

...and any combination of these (and much more)

String formatting - .format()

Python 3 introduced a new string formatting technique:

```
"Name: {}, Height: {}"
          .format("Barack Obama", 1.85)
```

output

```
'Name: Barack Obama, Height: 1.85'
```

Within the curly brackets, you can:

Specify a positional index:

```
"Name: {1}, Height: {0}".format(
    "Barack Obama", 1.85)
```

output

```
'Name: 1.85, Height: Barack Obama'
```

Conversion to specific type

```
"Name: {:s}, Height: {:f}".format(
    "Barack Obama", 1.85)
```

output

```
'Name: Barack Obama, Height: 1.850000'
```

...and any combination of these (and much more)

String formatting - .format()

Python 3 introduced a new string formatting technique:

```
"Name: {}, Height: {}"
          .format("Barack Obama", 1.85)
```

output

```
'Name: Barack Obama, Height: 1.85'
```

Within the curly brackets, you can:

Specify a positional index:

```
"Name: {1}, Height: {0}".format(
    "Barack Obama", 1.85)
```

output

```
'Name: 1.85, Height: Barack Obama'
```

Conversion to specific type

```
"Name: {:s}, Height: {:f}".format(
    "Barack Obama", 1.85)
```

output

```
'Name: Barack Obama, Height: 1.850000'
```

Refer by name

```
"Name: {name}, Height: {height}".format(
    name="Barack Obama", height=1.85)
```

output

```
'Name: Barack Obama, Height: 1.85'
```

...and any combination of these (and much more)

Types - Strings - formatting - Exercise

1. Create a string with your firstname, lastname and age, just like last week, but now using string formatting — and referring to the values *by name*. The output should look like this:

```
Name: myfirstname mylastname
Age: age
```

2. The `range` function creates a list of consecutive numbers. E.g.:

```
print(list(range(4)))
```

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

Try to explain what the following code does:

```
print(("%s\n"*5)%tuple(range(5)))
```

Types - Strings - formatting - Exercise - solution

1. Create a string with your firstname, lastname and age, just like before, but now using string formatting — and referring to the values *by name*. The output should look like this:

```
Name: Barack Obama
Age: 60
```
output

Types - Strings - formatting - Exercise - solution

1. Create a string with your firstname, lastname and age, just like before, but now using string formatting — and referring to the values *by name*. The output should look like this:

```
print("Name: %(fname)s %(lname)s \nAge: %(age)d" % {"fname": "Barack",
                                                    "lname": "Obama",
                                                    "age": 60})
```

or

```
print("Name: {fname} {lname} \nAge: {age:d}".format(fname="Barack",
                                                    lname="Obama",
                                                    age=60))
```

output

```
Name: Barack Obama
Age: 60
```

Types - Strings - formatting - Exercise - solution (2)

2. Try to explain what the following code does:

```
print(("%s\n"*5)%tuple(range(5)))
```

output

Types - Strings - formatting - Exercise - solution (2)

2. Try to explain what the following code does:

```
print(("%s\n"*5)%tuple(range(5)))
```

output

Types - Strings - formatting - Exercise - solution (2)

2. Try to explain what the following code does:

```
print(("%s\n"*5)%tuple(range(5)))
```

```
print("%s\n"*5)
```

output

Types - Strings - formatting - Exercise - solution (2)

2. Try to explain what the following code does:

```
print(("%s\n"*5)%tuple(range(5)))
```

```
print("%s\n"*5)
```

output

```
# Repeat string 5 times
'%s\n%s\n%s\n%s\n%s\n'
```

Types - Strings - formatting - Exercise - solution (2)

2. Try to explain what the following code does:

```
print(("%s\n"*5)%tuple(range(5)))
```

```
print("%s\n"*5)


print(tuple(range(5)))
```

output

```
# Repeat string 5 times
'%s\n%s\n%s\n%s\n%s\n'
```

Types - Strings - formatting - Exercise - solution (2)

2. Try to explain what the following code does:

```
print(("%s\n"*5)%tuple(range(5)))
```

```
print("%s\n"*5)


print(tuple(range(5)))
```

output

```
# Repeat string 5 times
'%s\n%s\n%s\n%s\n%s\n'


# Create [0,1,2,3,4] list and
# convert it to a tuple
(0, 1, 2, 3, 4)
```

Types - Strings - formatting - Exercise - solution (2)

2. Try to explain what the following code does:

```
print(("%s\n"*5)%tuple(range(5)))
```

```
print("%s\n"*5)



print(tuple(range(5)))



print('%s\n%s\n%s\n%s\n%s\n' %
    (0, 1, 2, 3, 4))
```

output

```
# Repeat string 5 times
'%s\n%s\n%s\n%s\n%s\n'


# Create [0,1,2,3,4] list and
# convert it to a tuple
(0, 1, 2, 3, 4)
```

Types - Strings - formatting - Exercise - solution (2)

2. Try to explain what the following code does:

```
print(("%s\n"*5)%tuple(range(5)))
```

```
print("%s\n"*5)



print(tuple(range(5)))



print('%s\n%s\n%s\n%s\n%s\n' %
    (0, 1, 2, 3, 4))
```

output

```
# Repeat string 5 times
'%s\n%s\n%s\n%s\n%s\n'



# Create [0,1,2,3,4] list and
# convert it to a tuple
(0, 1, 2, 3, 4)

# print numbers on separate lines
0
1
2
3
4
```

Conditionals

Conditionals

(also sometimes called "branching")

Branching - the if statement

The if statement is used to place a condition on a part of your code:

```
if condition:
    code block
```

Example:

```
if 2 < 5:
    print("Two is smaller than five")
```

output

```
Two is smaller than five
```

Branching - the if statement

The if statement is used to place a condition on a part of your code:

```
if condition:
    code block
```

Example:

```
if 2 < 5:
    print("Two is smaller than five")
```

output

```
Two is smaller than five
```

Note: This is the first (of many) examples where indentation is important to Python.

Branching - boolean expressions

`2 < 5` is a very simple truth expression. More complex expressions can be created using the boolean operators **and**, **or** and **not**:

```
if x > 30 and x < 60:
    print("value is within range")

if color == 'green' or color == 'red':
    color = 'yellow'

if name == 'Obama' and not (age < 20 or fav_dance == 'disco'):
    print("???")
```

Branching - the if-else statement

Often you will want to execute one piece of code when a condition is true, and another if is is false. This is done using the `if-else` statement:

```
if condition:
    code block
else:
    alternative code block
```

```
if 2 < 5:
    print("Two is smaller than five")
else:
    print("Two is larger than five")
```

output

```
Two is smaller than five
```

Branching - the if-else statement

Often you will want to execute one piece of code when a condition is true, and another if is is false. This is done using the `if-else` statement:

```
if condition:
    code block
else:
    alternative code block
```

```
if 2 < 5:
    print("Two is smaller than five")
else:
    print("Two is larger than five")
```

output

```
Two is smaller than five
```

Note that `if` and `else` must have the same indentation.

Branching - the if-elif-else statement

If you have several mutually exclusive conditions that you want to test for one at a time you can use the general *if-elif-else* version of the if statement:

```
if condition1:
    code block1
elif condition2:
    code block2
elif condition3:
    code block3
.
.
else:
    code block
```

Branching - the if-elif-else statement - example

Simulate a coin toss using a random number generator (RNG). We will import the module **random**, which implements an RNG:

```
import random
x = random.random()
if x >= 0 and x < 0.5:
    print("Heads")
elif x >= 0.5 and x < 1.0:
    print("Tails")
else:
    print("Number not a probability")
```

```
output
Heads
```

Branching - the if-elif-else statement - example

Simulate a coin toss using a random number generator (RNG). We will import the module **random**, which implements an RNG:

```
import random           # (explained next week)
x = random.random()
if x >= 0 and x < 0.5:
    print("Heads")
elif x >= 0.5 and x < 1.0:
    print("Tails")
else:
    print("Number not a probability")
```

output

```
Heads
```

Branching - the if-elif-else statement - example

Simulate a coin toss using a random number generator (RNG). We will import the module **random**, which implements an RNG:

```
import random           # (explained next week)
x = random.random()     # Random number between 0 and 1
if x >= 0 and x < 0.5:
    print("Heads")
elif x >= 0.5 and x < 1.0:
    print("Tails")
else:
    print("Number not a probability")
```

output

```
Heads
```

Branching - the if-elif-else statement - example

Simulate a coin toss using a random number generator (RNG). We will import the module **random**, which implements an RNG:

```
import random           # (explained next week)
x = random.random()     # Random number between 0 and 1
if x >= 0 and x < 0.5:      # Is x smaller than 0.5?
    print("Heads")
elif x >= 0.5 and x < 1.0:
    print("Tails")
else:
    print("Number not a probability")
```

output

```
Heads
```

Branching - the if-elif-else statement - example

Simulate a coin toss using a random number generator (RNG). We will import the module **random**, which implements an RNG:

```
import random           # (explained next week)
x = random.random()     # Random number between 0 and 1
if x >= 0 and x < 0.5:      # Is x smaller than 0.5?
    print("Heads")
elif x >= 0.5 and x < 1.0: # Otherwise, is x larger than 0.5?
    print("Tails")
else:
    print("Number not a probability")
```

output

```
Heads
```

Branching - the if-elif-else statement - example

Simulate a coin toss using a random number generator (RNG). We will import the module **random**, which implements an RNG:

```
import random           # (explained next week)
x = random.random()     # Random number between 0 and 1
if x >= 0 and x < 0.5:      # Is x smaller than 0.5?
    print("Heads")
elif x >= 0.5 and x < 1.0: # Otherwise, is x larger than 0.5?
    print("Tails")
else:
    print("Number not a probability")
```

output

```
Heads
```

Can we simplify this example?

Branching - the if-elif-else statement - example - simpler version

Original

```
import random
x = random.random()
if x >= 0 and x < 0.5:
    print("Heads")
elif x >= 0.5 and x < 1.0:
    print("Tails")
else:
    print("Number not a probability")
```

Branching - the if-elif-else statement - example - simpler version

Original

```
import random
x = random.random()
if x >= 0 and x < 0.5:
    print("Heads")
elif x >= 0.5 and x < 1.0:
    print("Tails")
else:
    print("Number not a probability")
```

Simplified

```
import random
x = random.random()
if x < 0.5:
    print("Heads")
else:
    print("Tails")
```

Branching - the if-elif-else statement - example - simpler version

Original

```
import random
x = random.random()
if x >= 0 and x < 0.5:
    print("Heads")
elif x >= 0.5 and x < 1.0:
    print("Tails")
else:
    print("Number not a probability")
```

Simplified

```
import random
x = random.random()     # Random number between 0 and 1
if x < 0.5:
    print("Heads")
else:
    print("Tails")
```

Branching - the if-elif-else statement - example - simpler version

Original

```
import random
x = random.random()
if x >= 0 and x < 0.5:
    print("Heads")
elif x >= 0.5 and x < 1.0:
    print("Tails")
else:
    print("Number not a probability")
```

Simplified

```
import random
x = random.random()     # Random number between 0 and 1
if x < 0.5:             # Is x smaller than 0.5?
    print("Heads")
else:
    print("Tails")
```

Branching - the if-elif-else statement - example - simpler version

Original

```
import random
x = random.random()
if x >= 0 and x < 0.5:
    print("Heads")
elif x >= 0.5 and x < 1.0:
    print("Tails")
else:
    print("Number not a probability")
```

Simplified

```
import random
x = random.random()     # Random number between 0 and 1
if x < 0.5:             # Is x smaller than 0.5?
    print("Heads")
else:                   # Otherwise, x must be larger than 0.5
    print("Tails")
```

Branches can be nested

```
if condition1:
    if condition2:
        code block2
    elif condition3:
        code block3
    else:
        code block4
```

Example:

```
if x>0
    if y>0:
        print("upper right quadrant")
    else:
        print("lower right quadrant")
else:
    if y>0:
        print("upper left quadrant")
    else:
        print("lower left quadrant")
```

Branching - exercise

1. Use the random number generator from one of the previous slides to generate a random number between 0 and 1, and use this number to simulate throwing a die.
2. Rides at amusement parks have height restrictions. We need to check if the user can go on a specific ride. The minimum height to get on the ride is 120cm, but somebody shorter than 130cm has to be accompanied by an adult. First, create a variable `height` and set it to an arbitrary value. Now write a program that display an appropriate message based on the value of this height variable. **Use nested branches.**
3. Same as previous exercise, but now using if-elif-else statement instead of nested branches

Branching - exercise - solution (1)

1. Use the random number generator from one of the previous slides to generate a random number between 0 and 1, and use this number to simulate throwing a die.

```
import random
x = random.random()
if x >= 0 and x < 1.0 / 6:
    print(1)
elif x >= 1.0 / 6 and x < 2.0 / 6:
    print(2)
elif x >= 2.0 / 6 and x < 3.0 / 6:
    print(3)
elif x >= 3.0 / 6 and x < 4.0 / 6:
    print(4)
elif x >= 4.0 / 6 and x < 5.0 / 6:
    print(5)
else:
    print(6)
```

Branching - exercise - solution (1)

1. Use the random number generator from one of the previous slides to generate a random number between 0 and 1, and use this number to simulate throwing a die.

```
import random
x = random.random()
if x >= 0 and x < 1.0 / 6:
    print(1)
elif x >= 1.0 / 6 and x < 2.0 / 6:
    print(2)
elif x >= 2.0 / 6 and x < 3.0 / 6:
    print(3)
elif x >= 3.0 / 6 and x < 4.0 / 6:
    print(4)
elif x >= 4.0 / 6 and x < 5.0 / 6:
    print(5)
else:
    print(6)
```

A more clever way to do this:

```
import random
x = random.random()
print(int(x * 6 + 1))
```

Branching - exercise - solution (2)

1.

```
height = 125
if height >= 120:
    if height >= 130:
        print('You can get on the ride')
    else:
        print('You need an adult to ride with you')
else:
    print('You are not tall enough to ride')
```

2.

```
height = 125
if height >= 130:
    print('You can get on the ride')
elif height >= 120:
    print('You need an adult to ride with you')
else:
    print('You are not tall enough to ride')
```

Loops

The while loop

A while loop repeats a piece of code for as long as its condition is true:

```
while condition:
      code block
```

```
i = 1
while (i < 4):
    print(i)
    i += 1

print("hello")
```

output

The while loop

A while loop repeats a piece of code for as long as its condition is true:

```
while condition:
     code block
```

```
i = 1               # i=1
while (i < 4):
    print(i)
    i += 1

print("hello")
```

output

The while loop

A while loop repeats a piece of code for as long as its condition is true:

```
while condition:
     code block
```

```
i = 1
while (i < 4):    # i=1    Check condition: True
    print(i)
    i += 1

print("hello")
```

output

The while loop

A while loop repeats a piece of code for as long as its condition is true:

```
while condition:
     code block
```

```
i = 1
while (i < 4):
    print(i)      # i=1    (note change in output)
    i += 1

print("hello")
```

output

```
1
```

The while loop

A while loop repeats a piece of code for as long as its condition is true:

```
while condition:
     code block
```

```
i = 1
while (i < 4):
    print(i)
    i += 1          # i=2

print("hello")
```

output

```
1
```

The while loop

A while loop repeats a piece of code for as long as its condition is true:

```
while condition:
     code block
```

```
i = 1
while (i < 4):    # i=2    Check condition: True
    print(i)
    i += 1

print("hello")
```

output

```
1
```

The while loop

A while loop repeats a piece of code for as long as its condition is true:

```
while condition:
     code block
```

```
i = 1
while (i < 4):
    print(i)      # i=2     (note change in output)
    i += 1

print("hello")
```

output

```
1
2
```

The while loop

A while loop repeats a piece of code for as long as its condition is true:

```
while condition:
     code block
```

```
i = 1
while (i < 4):
    print(i)
    i += 1        # i=3

print("hello")
```

output

```
1
2
```

The while loop

A while loop repeats a piece of code for as long as its condition is true:

```
while condition:
     code block
```

```
i = 1
while (i < 4):    # i=3    Check condition: True
    print(i)
    i += 1

print("hello")
```

output

```
1
2
```

The while loop

A while loop repeats a piece of code for as long as its condition is true:

```
while condition:
     code block
```

```
i = 1
while (i < 4):
    print(i)      # i=3    (note change in output)
    i += 1

print("hello")
```

output

```
1
2
3
```

The while loop

A while loop repeats a piece of code for as long as its condition is true:

```
while condition:
     code block
```

```
i = 1
while (i < 4):
    print(i)
    i += 1          # i=4

print("hello")
```

output

```
1
2
3
```

The while loop

A while loop repeats a piece of code for as long as its condition is true:

```
while condition:
     code block
```

```
i = 1
while (i < 4):    # i=4    Check condition: False
    print(i)
    i += 1

print("hello")
```

output

```
1
2
3
```

The while loop

A while loop repeats a piece of code for as long as its condition is true:

```
while condition:
     code block
```

```
i = 1
while (i < 4):
    print(i)
    i += 1

print("hello")     # i=4     (note change in output)
```

output

```
1
2
3
hello
```

while loop - exercise

1. Try out the program from the last slide, but leave out the i+=1. What happens?
2. Modify the program from the last slide to print the numbers in reverse order.
3. Optional: Modify the program from the last slide to print out all the numbers between 1 and 10 which are not divisible by three.

while loop - exercise - solution

1. Try out the program from the last slide, but leave out the i+=1. What happens?

2. Modify the program from the last slide to print the numbers in reverse order.

```
i = 3
while (i > 0):
    print(i)
    i -= 1
```

while loop - exercise - solution

1. Try out the program from the last slide, but leave out the i+=1. What happens?
   It never ends. This is called an *infinite loop*.
2. Modify the program from the last slide to print the numbers in reverse order.

```
i = 3
while (i > 0):
    print(i)
    i -= 1
```

while loop - exercise - solution (2)

3. Modify the program from the last slide to print out all the numbers between 1 and 10 which are not divisible by three.

```
i = 0
while (i < 10):
    if i%3 != 0:        # % = modulo operator = remainder in division
        print(i)
    i += 1
```

The for loop

Most of the time when writing a loop, it will be to iterate through a string, list or tuple. The for command makes this easy:

```
for variable in container-object:
     code block
```

```
container = ["a", "b", "c"]
for val in container:
    print(val)
```

output

The for loop

Most of the time when writing a loop, it will be to iterate through a string, list or tuple. The for command makes this easy:

```
for variable in container-object:
     code block
```

```
container = ["a", "b", "c"]
for val in container:    # val = "a"
    print(val)
```

output

The for loop

Most of the time when writing a loop, it will be to iterate through a string, list or tuple. The for command makes this easy:

```
for variable in container-object:
     code block
```

```
container = ["a", "b", "c"]
for val in container:
    print(val)           # val = "a"  (note change in output)
```

output

```
a
```

The for loop

Most of the time when writing a loop, it will be to iterate through a string, list or tuple. The for command makes this easy:

```
for variable in container-object:
     code block
```

```
container = ["a", "b", "c"]
for val in container:    # val = "b"
    print(val)
```

output

```
a
```

The for loop

Most of the time when writing a loop, it will be to iterate through a string, list or tuple. The for command makes this easy:

```
for variable in container-object:
     code block
```

```
container = ["a", "b", "c"]
for val in container:
    print(val)           # val = "b"  (note change in output)
```

output

```
a
b
```

The for loop

Most of the time when writing a loop, it will be to iterate through a string, list or tuple. The for command makes this easy:

```
for variable in container-object:
     code block
```

```
container = ["a", "b", "c"]
for val in container:    # val = "c"
    print(val)
```

output

```
a
b
```

The for loop

Most of the time when writing a loop, it will be to iterate through a string, list or tuple. The for command makes this easy:

```
for variable in container-object:
     code block
```

```
container = ["a", "b", "c"]
for val in container:
    print(val)          # val = "c"  (note change in output)
```

output

```
a
b
c
```

The range function

You will also often find yourself iterating through a range of numbers. The range function allows you to create a list of numbers running from start to stop-1 with some step size.

```
range([start], stop, [step-size])
```

You can then iterate through this list using the for statement:

```
for i in range(3):          # range(3) -> [0,1,2]
    print(i+1)
```

output

```
1
2
3
```

The range function

You will also often find yourself iterating through a range of numbers. The range function allows you to create a list of numbers running from start to stop-1 with some step size.

```
range([start], stop, [step-size])
```

You can then iterate through this list using the for statement:

```
for i in range(3):          # range(3) -> [0,1,2]
    print(i+1)
```

output

```
1
2
3
```

Note that start and step-size are optional parameters.

The for loop - with indices

If we need to know the index, we can iterate over indices instead:

Original

```
container = ["a", "b", "c"]
for val in container:
    # I don't know the index
    print(val)
```

Iterating over indices

```
container = ["a", "b", "c"]
for i in range(len(container)):
    # Now I know the current index i
    print(container[i])
```

The for loop - with indices

If we need to know the index, we can iterate over indices instead:

Original

```
container = ["a", "b", "c"]
for val in container:
    # I don't know the index
    print(val)
```

Iterating over indices

```
container = ["a", "b", "c"]
for i in range(len(container)):
    # Now I know the current index i
    print(container[i])
```

There is a built-in function `enumerate`, that make this easier:

```
container = ["a", "b", "c"]
for i,val in enumerate(container):          # Note the i,val tuple after for
    # Now I know the current index i
    print(val)
```

The for loop - with indices

If we need to know the index, we can iterate over indices instead:

Original

```
container = ["a", "b", "c"]
for val in container:
    # I don't know the index
    print(val)
```

Iterating over indices

```
container = ["a", "b", "c"]
for i in range(len(container)):
    # Now I know the current index i
    print(container[i])
```

There is a built-in function `enumerate`, that make this easier:

```
container = ["a", "b", "c"]
for i,val in enumerate(container):          # Note the i,val tuple after for
    # Now I know the current index i
    print(val)
```

How does this work? Enumerate creates a list of 2-tuples

```
print(list(enumerate(container)))
```

```
[(0, 'a'), (1, 'b'), (2, 'c')]
```
output

Interrupting a loop

There are several ways to interrupt a loop from within:

- **break** - stop loop immediately
- **continue** - immediately start on the next iteration.

Interrupting a loop

There are several ways to interrupt a loop from within:

- **break** - stop loop immediately
- **continue** - immediately start on the next iteration.

```
for i in range(1,5):
    if i==3:
        continue  # start on next iteration (skips print)
    print(i)
```

output

```
1
2
4
```

Interrupting a loop

There are several ways to interrupt a loop from within:

- **break** - stop loop immediately
- **continue** - immediately start on the next iteration.

```
for i in range(1,5):
    if i==3:
        continue  # start on next iteration (skips print)
    print(i)
```

output

```
1
2
4
```

What would happen if I wrote `break` instead of `continue`?

Interrupting a loop

There are several ways to interrupt a loop from within:

- **break** - stop loop immediately
- **continue** - immediately start on the next iteration.

```
for i in range(1,5):
    if i==3:
        continue  # start on next iteration (skips print)
    print(i)
```

output

```
1
2
4
```

What would happen if I wrote `break` instead of `continue`?

`break` and `continue` work for both while and for loops.

Loops - else statement

Loops can have an else clause, which will execute if the loop finishes normally (without a break statement).

```
>>> l = range(1, 10)
>>> for i in l:
...     if i == 12:
...         break
... else:
...     print(" Element not found ")
...
Element not found
```

Nice to know: the pass statement

Since Python relies on correct indentation, you have to write *something* inside an `if`-statement or loop

Nice to know: the pass statement

Since Python relies on correct indentation, you have to write *something* inside an `if`-statement or loop

Sometimes, you just want to do nothing.

Nice to know: the pass statement

Since Python relies on correct indentation, you have to write *something* inside an `if`-statement or loop

Sometimes, you just want to do nothing.

There is a statement that can be used as a placeholder, since it does literally nothing:

```
if x < 0:
     pass    # Haven't decided what to do here yet
elif x > 0:
     print("positive")
```

Nested loops

Nested data structures often require nested loops:

```
c = [(0,0,1), (0,2,1), (5,2,4), (3,6,4)]
for tup in c:                   # Iterate over outer list
    for entry in tup:           # Iterate over inner tuple
        print(entry)
```

```
output
0
0
1
0
2
1
5
2
4
3
6
4
```

Nested loops - continued

```
presidents = [{'name':'Barack Obama', 'age':56},
              {'name':'Bill Clinton', 'age':71}]
for dictionary in presidents:
    for key in dictionary.keys():
        print("%s: %s" % (key, dictionary[key]))
```

output

```
age: 56
name: Barack Obama
age: 71
name: Bill Clinton
```

Nested loops - continued

```
presidents = [{'name':'Barack Obama', 'age':56},
              {'name':'Bill Clinton', 'age':71}]
for dictionary in presidents:
    for key in dictionary.keys():
        print("%s: %s" % (key, dictionary[key]))
```

output

```
age: 56
name: Barack Obama
age: 71
name: Bill Clinton
```

Btw: note (again) that order is not preserved in a dictionary (in python < 3.7)

Loops - exercise

1. Create the following list: [6,9,4,8,7,1,2].
2. Write a while loop that prints each element in the list.
3. Write a for loop that prints each element in the list.
4. Write a loop that prints all elements that are larger than their left neighbour (for the above example: 9, 8, 2). Which type of loop is best suited for this purpose?

Loops - exercise - solution

1. Create the following list: [6,9,4,8,7,1,2].

```
number_list = [6,9,4,8,7,1,2]
```

2. Write a while loop that prints each element in the list.

```
i = 0
while i < len(number_list):
    print(number_list[i])
    i += 1
```

3. Write a for loop that prints each element in the list.

```
for entry in number_list:
    print(entry)
```

Loops - exercise - solution (2)

4. Write a loop that prints all elements that are larger tha
   their left neighbour (for the above example: 9, 8, 2).
   Which type of loop is best suited for this purpose?

```
for i in range(1, len(number_list)):          # i starts at one
    if number_list[i] > number_list[i-1]:     # compare to left-neighbor
        print(number_list[i])
```

Loops - exercise - solution (2)

4. Write a loop that prints all elements that are larger tha
   their left neighbour (for the above example: 9, 8, 2).
   Which type of loop is best suited for this purpose?

```
for i in range(1, len(number_list)):          # i starts at one
    if number_list[i] > number_list[i-1]:     # compare to left-neighbor
        print(number_list[i])
```

Alternatively, there is a way of automatically getting bo
value and index:

```
for i,entry in enumerate(number_list): # enumerate returns (i,value) pa
    if i > 0 and entry > number_list[i-1]:
        print(entry)
```

Loops - exercise - solution (2)

4. Write a loop that prints all elements that are larger tha
   their left neighbour (for the above example: 9, 8, 2).
   Which type of loop is best suited for this purpose?

```
for i in range(1, len(number_list)):          # i starts at one
    if number_list[i] > number_list[i-1]:     # compare to left-neighbor
        print(number_list[i])
```

   Alternatively, there is a way of automatically getting bo
   value and index:

```
for i,entry in enumerate(number_list): # enumerate returns (i,value) pa
    if i > 0 and entry > number_list[i-1]:
        print(entry)
```

Moral of the story: even when you need indices, its still easier to use `for` than `while`.