Exercise: The basics - variables and types

Aim: Introduce python variables and types.

Issues covered:

- Using the python interactive shell
- Creating variables
- Using “print” to display a variable
- Simple arithmetic
- Converting between types
- Single-line comments
- White-space makes code readable
- In the python interactive shell you don’t need “print”

1. Let's start using the Python interactive shell and create some variables.

a. Type "`python`" at the command-line to start the python interactive shell.
b. Create a variable called "`course`" and assign it the value "`python`".
c. Create a variable called "`rating`" and assign it an integer value (anything you like).
d. Print both variables to the screen using the "`print`" command.

2. Let's use Pythagoras theorem to calculate the length of the hypotenuse of a right-angled triangle where the other sides have lengths 3 and 4.

a. Create variables called "b" and "c" and assign them the values of the sides with known lengths.
b. Write a mathematical expression to calculate the length of the hypotenuse (REMEMBER: “a-squared equals b-squared plus c-squared").
c. Create a variable "a" that equals the result of the calculation.
d. Print the value of variable "a".

3. Let's take a look at some data types.

a. Print the data type of each of the variables: `a, b, c`
b. Can you work out why "a" is different from "b" and "c"?

4. Let's investigate converting between data types.

a. Use the "`int()`" built-in function to convert "a" into an integer type.
b. Print the data type of "a" to check it worked.
c. Now try running the following line of code: `print a + " squared equals " + b + " squared " + c + " squared."`
d. You should get an error, do you know why?
e. Try using the built-in "`str()`" (convert to string) function to fix the error.

5. It is easy to write illegible code.

Let's start some good habits today. Adding comment lines at regular intervals helps to break up your code and make it easier to understand. Python does not enforce use of white-space around operators (such as "=") but it really helps the reader. Please compare the readability of these two code blocks:

```
num=24
a=34.999
result=num*(13-a**2)+1.0
print "Result:",result
```

```
# Set up calculation inputs
num = 24
a = 34.999

# Calculate result using the Godzilla algorithm
result = num * (13 - a**2) + 1.0
print "Result:", result
```

Solution: The basics - variables and types

1.

```
$ python
>>> course = "python"
>>> rating = 3
>>> print course, rating
>>> # In the interactive shell you don't need "print"
>>> course, rating  # is the same as:  print course, rating
```

2.

```
>>> b = 3
>>> c = 4
>>> (b**2 + c**2)**0.5
>>> a = (b**2 + c**2)**0.5
>>> print a
```

3.

```
>>> print type(a)
<type 'float'>
>>> print type(b), type(c)
<type 'int'> <type 'int'>
```

4.

```
>>> a = int(a)
>>> print type(a)

>>> print a + " squared equals " + b + " squared " + c + " squared."

Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    print a + " squared equals " + b + " squared " + c + " squared."
TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> print str(a) + " squared equals " + str(b) + " squared " + str(c) +   "
      squared."
5 squared equals 3 squared 4 squared.
```

Exercise: Control flow

Aim: Introduce Python syntax for loops and if statements.

Issues covered:

- while statement (and escaping loops)
- if/elif/else statements
- for statement
- Indentation
- Testing for truth

1. Let's create some "while" loops.

a. Create a never-ending "`while`" loop; use 4 spaces for indentation.
b. How do you escape this?
c. Create a "`while`" loop that never gets called.

2. Let's step through the items in a list using the "for" statement.

a. Create a variable called "`gases`" assigned the list: `['He', 'Ne', 'Ar', 'Kr']`
b. Create a counter variable "`count`" assigned the value of 0.
c. Start a "`while`" loop that will loop while "`count`" is less than 4.
d. Within the "`while`" loop:
    - Assign each value in "`gases`" to a temporary variable called "`item`".
    - Print each value of "`item`", along with its position in the list ("`count`").
    - Add `1` to the value of "`count`". (What would happen if you missed this part?)

3. Let's try out "if"/"elif"/"else".

a. Create a variable called "`name`" assigned the value `"Lisa"`.
b. Write an `"if"` block that tests if `"name"` is `"Lisa":` if True then print the name and `"plays saxophone".`
c. Write an `"elif"` block that tests if `"name"` is `"Bart":` if True then print the name and `"rides skateboard".`
d. Write an `"else"` block: if True then print the name and `"lives in Springfield".`

4. Let's test the "truth" of different objects. In most programming languages any object can be tested for truth value, using an "if" or "while" condition.

a. Create the variable "`x`" and assign the value `1` to it.
b. Using the following code to test if `x` is true: `if x: print x, " is True"`
c. Test if the following values are `"True": 22.1, “hello”, [1, 2].`
d. Now try testing for truth with zero (integer) and zero (float): `0, 0.0`
e. Try some values that are likely to be false: `None, False`
f. Now try an empty string, empty sequences and a dictionary: `"" , [], {}, ()`
NOTE: It is useful to remember that zero and empty objects are not considered true!

Solution: Control flow

1.

```
>>> while 1:
        x = 1 # But it's of little significance
        # Press Ctrl+C to escape the loop
>>> while 0:
        pass
```

2.

```
>>> gases = ['He', 'Ne', 'Ar', 'Kr']
>>> count = 0
>>> while count < 4:
        item = gases[count]
        print item, count
        count += 1
```

3.

```
>>> name = 'Lisa'
>>> if name == 'Lisa':
        print name, 'plays saxophone'
    elif name == 'Bart':
        print name, 'rides skateboard'
    else:
        print name, 'lives in Springfield'
```

4.

```
>>> x = 1
>>> if x: print x, " is True"
1 is True

>>> if 22.1: print "True"
True

>>> if "hello": print "True"
True

>>> if [1, 2]: print "True"
True
```

```
>>> if 0: print "True"

>>> if 0.0: print "True"

>>> if None: print "True"

>>> if False: print "True"

>>> if "": print "True"

>>> if []: print "True"

>>> if {}: print "True"

>>> if (): print "True"
```

Exercise: Lists and Slicing

Aim: Introduce lists and their methods.

Issues covered:

- Creating a list
- Indexing and slicing lists
- Using methods on lists
- Using built-in dir() and help() functions to interrogate objects
- Reading from standard input

1. Let's create a simple list and play with it.

a. Create a list "x" containing integers: `1, 2, 3, 4, 5`
b. Use list indexing to select the second item in the list.
c. Use list indexing to select the second to last item in the list.
d. Use list slicing to select the whole list.
e. Use list slicing to select the second to the fourth item (inclusive) in the list.

2. Let's create a list from a range and play with it.

a. Create a list "y" with values from 1 to 10 (use the "`range()`" function).
b. Replace the first item in the list with the value 10.
c. Append the number 11 to the list.
d. Add another list (`[12, 13, 14]`) to the list using the "`extend`" method.

3. Let’s combine lists and loops and explore some list methods.

a. Create two empty lists called "`forward`" and "`backward`".
b. Create a variable called "`values`" and assign it the list: `["a", "b", "c"]`
c. Create a "`for`" block to loop over each item in "`values`".
d. Inside the loop:
    - Append each value to the list called "`forward`".
    - Insert the same value to the front of the list called "`backward`".
e. On exiting the loop print the values of "`forward`" and "`backward`".
f. Reverse the order of "`forward`".
g. Check that "`forward`" and "`backward`" are the same.

4. Let's find out what else you can do with a list.

a. Create a list named "`countries`" as follows: `["uk", "usa", "uk", "uae"]`
b. Use "`dir(countries)`" to display the properties and methods of the list.
c. Print the documentation on the "count" method of the list using "`help(countries.count)`".
d. Now use that method to count the number of times "`uk`" appears in the list "`countries`".

Solution: Lists and Slicing

1.

```
>>> x = [1, 2, 3, 4, 5]
>>> print x[1]
>>> print x[-2]
>>> print x[:]
>>> print x[1:4]
```

2.

```
>>> y= range(1, 11)
>>> y[0] = 10
>>> y.append(11)
>>> y.extend([12, 13, 14])
>>> print y
```

3.

```
>>> forward = []
>>> backward = []
>>> values = ["a", "b", "c"]

>>> for item in values:
        forward.append(item)
        backward.insert(0, item)

>>> print "Forward is:", forward
>>> print "Backward is:", backward
>>> forward.reverse()
# Or you could use slicing to reverse: forward = forward[::-1]
>>> print forward == backward
```

4.

```
>>> countries = ["uk", "usa", "uk", "uae"]
>>> dir(countries)  # Output not shown here
>>> help(countries.count) # Shows documentation about the "count" method (shown
below)
Help on built-in function count:
count(...)
    L.count(value) -> integer -- return number of occurrences of value

>>> # You can use dir() and help() on any object in python
>>> countries.count("uk")
```

Exercise: Tuples

Aim: Introduce tuples and how to work with them.

Issues covered:

- Working with tuples
- Converting a list to a tuple
- The enumerate built-in function
- Swapping values with tuples

1. Let's create a couple of tuples.

a. Create a tuple named "`t`" containing one integer 1.
b. Use indexing to print the last value of the tuple.
c. Create a tuple containing all values from 100 to 200. There must be a short-cut to doing this...there is a built-in function to convert to a tuple called: `tuple()`
d. Print the first and the last items in the tuple.

2. Let's use the "enumerate" built-in function. In an earlier exercise we stepped through the items in a list using the "for" statement. We also wanted to print the index (count) of each item in the list. The solution was:

```
>>> mylist = [23, "hi", 2.4e-10]
>>> for item in mylist:
        print item, mylist.index(item)
```

a. Re-write the "`for`" loop above using the "`enumerate()`" built-in function.
b. For each value assign the tuple: `(count, item)`

3. Let's unpack multiple values from a list to a tuple in one line.

a. Assign the new variables "`first`", "`middle`" and "`last`" to the values of "`mylist`" created above.
b. Print the values of "`first`", "`middle`" and "`last`".
c. Now re-assign the values of variables "`first`", "`middle`" and "`last`" to the values of "`middle`", "`last`" and "`first`" - in one line!

Solution: Tuples

1.
```
>>> t = (1,)
>>> print t[-1]

>>> l = range(100, 201)
>>> tup = tuple(l)
>>> print tup[0], tup[-1]
```

2.
```
>>> mylist = [23, "hi", 2.4e-10]
>>> for (count, item) in enumerate(mylist):
        print count, item
```

3.
```
>>> (first, middle, last) = mylist
>>> print first, middle, last
>>> (first, middle, last) = (middle, last, first)
>>> print first, middle, last
```

Exercise: Input and Output

Aim: Introduce reading and writing from files.

Issues covered:

- Opening files to read or write to
- Reading data in blocks, lines or all
- Writing data to files
- Working with binary files

1. Let's read the entire contents of the CSV file and display each line.

a. Copy the file "`example_data/weather.csv`" to the current directory.
b. Use the "`with`" statement to open the file (in read mode, "`r`") "`weather.csv`".
c. Read the contents of the file into the variable called "`data`" using the "`read`" method.
d. Print the contents of “`data`”.

2. Let's try reading the file line by line.

a. Use the "`with`" statement to open the file "`weather.csv`".
b. Read the first line using the "`readline()`" method of the file handle.
c. Start a while loop that continues until "`readline()`" returns an empty string.
d. Whilst inside the while loop read each line using the "`readline()`" method.
e. Print each line that is read.
f. When the loop has exited print "`It's over`".

3. Let's do the same thing using a "for" loop and grab just the rainfall values.

a. Use the "`with`" statement to open the file "`weather.csv`".
b. Read the first line using the "`readline()`" method of the file handle.
c. Create an empty list called "`rain`".
d. Create a "`for`" loop that reads each line as a variable "`line`". Print each line within the loop.
e. Can you extract only the rainfall values from the final column, convert them to real types (decimals) using "`float`" and append them to the list "`rain`"?
f. Print the contents of "`rain`".
g. Now try writing the contents of rain to a file called "`myrain.txt`". (Use the "`write()`" method of a file handle).

4. Let's try writing and reading some binary data [ADVANCED].

a. Firstly, we'll need to import the "`struct`" module which allows us to pack/unpack data to/from binary: `import struct`
b. Pack the following list of values into four bytes using the "`struct.pack`" function: `bin_data = struct.pack("bbbb", 123,12,45,34)`
c. Use the "`with`" statement to create a binary file handle in write mode ("`wb`") to a file called "`mybinary.dat`".
d. Write the binary data to the file.
e. Use the "`with`" statement to open the file (in binary read mode).
f. Read the contents of the file into a variable called "`bin_data2`".
g. Unpack the binary data using: `data = struct.unpack("bbbb", bin_data2)`
h. Print the "`data`" variable to check it contains the same four values you started with.

Solution: Input and Output

1.
```
>>> with open("weather.csv", "r") as reader: # can omit the "r" when reading
        data = reader.read()
>>> print data
```

2.
```
>>> with open("weather.csv") as reader:
        line = reader.readline()
        while line:
            print line
            line = reader.readline()

>>> print "It's over"
```

3.
```
>>> with open("weather.csv") as reader:
        header = reader.readline() #   We will ignore this
        rain = []
        for line in reader.readlines():
            r = line.strip().split(",")[-1]
            r = float(r)
            rain.append(r)

>>> print rain
>>> with open("myrain.txt", "w") as writer:
        for r in rain:
            writer.write(str(r) + "\n")
```

4.
```
>>> import struct
>>> bin_data = struct.pack("bbbb", 123, 12, 45, 34)

>>> with open("mybinary.dat", "wb") as bwriter:
        bwriter.write(bin_data)

>>> with open("mybinary.dat", "rb") as breader:
        bin_data2 = breader.read()

>>> data = struct.unpack("bbbb", bin_data2)
>>> print data
```

Exercise: Strings

Aim: Introduce strings, their usage and methods

Issues covered:

- Creating strings
- Splitting and selecting from strings
- String methods
- Converting to/from strings

1. Let's loop through a string as a sequence (like a list, tuple).

a. Create a string "s" with the value "`I love to write python`".
b. Loop through "s" displaying each value.
c. Print the 5th element of "s".
d. Print the last element of "s". (Remember, you can use a negative index).
e. Print the length of "s".
f. Try printing: `s[0], s[0][0], s[0][0][0]`. Can you explain what is going on?

2. Let's try splitting a string to loop through its words (rather than characters).

a. Create a string "s" with the value "`I love to write python`".
b. Split the string into a list of words called "`split_s`".
c. Loop through the list of words; If any of the words contain the letter "`i`" print "`I found 'i' in: '<WORD>'`" (where <WORD> is the word).
d. Note: What is the difference between '`word.find("i")`' and '`word.find("i") > -1`'? What does it return if “`i`” is not found?

3. Let's explore other useful aspects of strings.

a. Create a string called "`something`" with the value "`Completely Different`".
b. Print all the properties and methods of the string.
c. Use a string method to count the number of times the character "`t`" is present in the string.
d. Use a string method to find the first position of the sub-string "`plete`" in the string.
e. Use a string method to split the string by the character "`e`".
f. Create a new string ("`thing2`") that replaces the word "`Different`" with "`Silly`".
g. Try to assign the letter "`B`" to the first character of the string "`something`" using: `something[0] = "B"`. (Why does this cause an error?)

Solution: Strings

1.

```
>>> s = "I love to write python"
>>> for i in s:
        print i

>>> print s[4]
>>> print s[-1]
>>> print len(s)

>>> print s[0]
>>> print s[0][0]
>>> print s[0][0][0]
```

2.

```
>>> s = "I love to write python"
>>> split_s = s.split()

>>> print split_s
>>> for word in split_s:
      if word.find("i") > -1:
           print "I found 'i' in: '{0}'".format(word)  # ({0} not {} in Py2.6)

>>> # NOTE: You can also do this to find a sub-string in a string
>>> if "i" in word:
        print "I found 'i' in '{0}'".format(word)
```

3.

```
>>> something = "Completely Different"
>>> print dir(something)

>>> something.count("t")
>>> something.find("plete")

>>> something.split("e")
>>> thing2 = something.replace("Different", "Silly")
>>> print thing2

>>> something[0] = "B"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment

>>> # Strings are immutable – they cannot be changed!
```

Exercise: Aliasing

Aim: Introduce how Python uses aliases

Issues covered:

- Testing for aliases
- Modifying target objects through aliases
- Avoiding aliasing using “deepcopy”

1. Let's create an alias and try changing the original variable and the alias.

a. Create a list "a" with the value [0, 1, 2].
b. Create a variable "b" and assign it the value variable "a".
c. Print "a" and "b".
d. Modify "b" so that its first member is “hello”.
e. Print "a" and "b".
f. Append the value 3 to list "a".
g. Print "a" and "b".

2. Let's try it with a string.

a. Create a string "a" with the value "can I change".
b. Create a variable "b" and assign it the value variable "a".
c. Print "a" and "b".
d. Set the value of "b" to “different”.
e. Print "a" and "b".
f. What is different about lists and strings that causes this behaviour?

3. When we want to avoid aliasing we can force a “deep” copy.

a. Create a list "a" with the value [0, 1, 2].
b. Create a variable "b" and assign it a deep copy of variable "a" (use: `copy.deepcopy`).
c. Print "a" and "b".
d. Modify "b" so that its first member is “hello”.
e. Print "a" and "b".

Solution: Aliasing

1.

```
>>> a = range(3)
>>> b = a
>>> print a, b
[0, 1, 2] [0, 1, 2]
>>> b[0] = "hello"
>>> print a, b
['hello', 1, 2] ['hello', 1, 2]
>>> a.append(3)
>>> print a, b
['hello', 1, 2, 3] ['hello', 1, 2, 3]
```

2.

```
>>> a = "can I change"
>>> b = a
>>> print a, b
can I change can I change
>>> b = "different"
>>> print a, b
can I change different
```

3.

```
>>> import copy
>>> a = range(3)
>>> b = copy.deepcopy(a)
>>> print a, b
[0, 1, 2] [0, 1, 2]
>>> b[0] = "hello"
>>> print a, b
[0, 1, 2] ['hello', 1, 2]
```

Exercise: Functions

Aim: Introduce writing and calling functions.

Issues covered:

- Writing a simple function
- Indenting code within a function
- Sending arguments to functions
- Calling functions
- Checking arguments in functions

1. Let's create a simple function.

a. Use the "`def`" statement to define a function called "`double_it`".
b. The function should take one argument called "`number`".
c. It should return double the "`number`" argument.
d. Test the function by calling it with an integer and then a float.
e. What happens if you send it a string rather than a number?

2. Let's write a simple function to calculate the length of the hypotenuse of a right-angled triangle given the length of the other sides (called "a" and "b").

a. Use the "`def`" statement to define a function called "`calc_hypo`" that takes two arguments: "`a`" and "`b`".
b. Inside the function, calculate the length of the hypotenuse and assign it to a variable called "`hypo`".
c. Return the value of "`hypo`".
d. Test out the function by calling it with values (`3, 4`).

3. Let's improve the function above by adding some checks into the code.

a. Add a check on the arguments to make sure are of type "`float`" or "`int`".
b. Add a check on the arguments to make sure they are greater than zero.
c. If any argument fails the checks: `print "Bad argument"` and return `False`.
d. Call the function with different arguments to test that the checks are working.

Solution: Functions

1.
```
>>> def double_it(number):
        return 2 * number

>>> double_it(2)
4
>>> double_it(3.5)
7.0
>>> double_it("hello")
'hellohello'
```

2.
```
>>> def calc_hypo(a, b):
        hypo = (a**2 + b**2)**0.5
        return hypo

>>> print calc_hypo(3, 4)
5.0
```

3.
```
>>> def calc_hypo(a, b):
        if type(a) not in (int, float) or type(b) not in (int, float):
            print "Bad argument"
            return False
        if a <= 0 or b <= 0:
            print "Bad argument"
            return False
        hypo = (a**2 + b**2)**0.5
        return hypo

>>> calc_hypo(0, -2)
>>> calc_hypo("hi", "bye")
```

Exercise: Scripts and Libraries

Aim: Introduce scripts and importing from libraries

Issues covered:

- Become familiar with running a python script
- Working with command line arguments
- Create your own library
- Learn about the `"__init__.py"` module
- Use a script to import your own library to call a function

1. Let's make a python library called “dancing”.

a. Create a directory (in the Linux shell) called "`dancing`".
b. Test if you can import the library by running: `python -c "import dancing"`
   (Does this work?)
c. Create an empty file inside the "`dancing`" directory called `"__init__.py"`.
d. Test if you can import the library by running: `python -c "import dancing"`
   (Does it work this time?)

2. Let's create the "dance.py" module inside our library.

a. Copy the file "`example_code/dance.py`" to the “`dancing`” directory.
b. Have a look inside the "`dancing/dance.py`" module. Do you understand what should happen in the "boogie" function?
c. Test if you can import the module by running: `python -c "import dancing.dance"`
d. Test if you can import the "`boogie`" function from the module by running: `python -c "from dancing.dance import boogie"`

3. Let's write a 4 line a script that allows users to interact with our library.

a. Create a script in the current directory (not in "`dancing`") called "`dancer.py`", containing lines to:
   L1: Import the "`sys`" module
   L2: Import the "`boogie`" function from the "`dancing.dance`" module
   L3: Set a variable "`moves`" to all command-line arguments sent to the script ("`sys.argv`")
   L4: Call the "`boogie`" function with the variable "`moves`".
b. Test running the script without any arguments.
c. Test running the script with some of your own dance moves (as command-line arguments).

Solution: Scripts and Libraries

1.

```
Linux: $ mkdir dancing
Linux: $ python –c "import dancing"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named dancing

Linux: $ touch dancing/__init__.py    # "touch" creates a file if not there.
Linux: $ python –c "import dancing"
```

2.

```
Linux: $ cp example_code/dance.py dancing/

Linux: $ python –c "import dancing.dance"
Linux: $ python –c "from dancing.dance import boogie"
```

3.

```
Suitable contents of dancer.py script:

     import sys
     from dancing.dance import boogie
     moves = sys.argv[1:]
     boogie(moves)

Linux: $ python dancer.py

Linux: $ python dancer.py Twist Watusi Headbang
```

Exercise: Sets and Dictionaries

Aim: To start working with sets and dictionaries.

Issues covered:

- Creating and using sets
- Creating dictionaries
- Working with dictionaries
- Dictionary methods

1. Let's create two sets and work with them.

a. Create a set called "a" containing values: `[0, 1, 2, 3, 4, 5]`
b. Create a set called "b" containing values: `[2, 4, 6, 8]`
c. Print the union of the two sets.
d. Print the intersection of the two sets.

2. Let’s use a collect up counts using a dictionary.

a. Create a list "`band`" with the members: `["mel", "geri", "victoria", "mel", "emma"]`
b. Create an empty dictionary called "`counts`".
c. Loop through each member of "`band`".
d. Inside the loop: when a name is seen for the first time add an item to the dictionary with the name as the key and 1 as the value.
e. Inside the loop: if a name has already been seen then add 1 to the dictionary item to indicate that it has been spotted again. The dictionary is storing a count of each name.
f. Loop through the dictionary printing the key and value of each item.

3. Let's look at some other useful characteristics of dictionaries.

a. What happens if you test the truth value of the empty dictionary? `if {}: print 'hi'`
b. Create a dictionary "d" as follows: `{"maggie": "uk", "ronnie": "usa"}`
c. Take a look at the properties and methods of a dictionary with "`dir(d)`".
d. Try out the following methods: `d.items(), d.keys(), d.values()`
e. Can you work out how to use the "`d.get()`" method to send a default value if the key does not exist?
f. What about "`d.setdefault()`"? It's a useful shortcut.

Solution: Sets and Dictionaries

1.

```
>>> a = set([0, 1, 2, 3, 4, 5])
>>> b = set([2, 4, 6, 8])
>>> print a.union(b)
set([0, 1, 2, 3, 4, 5, 6, 8])
>>> print a.intersection(b)
set([2, 4])
```

2.

```
>>> band = ["mel", "geri", "victoria", "mel", "emma"]
>>> counts = {}

>>> for name in band:
        if name not in counts:
            counts[name] = 1
        else:
            counts[name] += 1

>>> for name in counts:
        print name, counts[name]
```

3.

```
>>> if {}: print 'hi'  # is not True

>>> d = {"maggie": "uk", "ronnie": "usa"}
>>> dir(d)
['__class__', ..., 'viewvalues']
>>> print d.items()
[('maggie', 'uk'), ('ronnie', 'usa')]
>>> print d.keys()
['maggie', 'ronnie']
>>> print d.values()
['uk', 'usa']
>>> d.get("maggie", "nowhere")
'uk'
>>> d.get("ringo", "nowhere")
'nowhere'
>>> res = d.setdefault("mikhail", "ussr")
>>> print res, d["mikhail"]
ussr ussr
```

Exercise: Object Oriented Programming (OOP)

Aim: Introduce OOP: classes, instances, methods and attributes.

Issues covered:

- Basic OOP terminology
- Define your own class
- Use the `__init__()` constructor method
- Define your own methods
- Define and modify attributes within classes
- Interacting with instances of your class (objects)
- Encapsulating the data and the functions in a single class

1. Let's use an existing class called "Band" (for band managers).

a. Copy the file "`example_code/band.py`" to the current directory.
b. Test if you can import the module by running: `python –c "import band"`
   (If python raises an Exception what can you do to fix it?)
c. Read through the code in "`band.py`" so that you understand how it works.

2. Let's manage our first band.

a. Start an interactive Python session and import the `Band` class from "`band.py`".
b. Create a variable "`ws`" as an instance of Band named "`The White Stripes`".
c. Work with the instance ("`ws`") to employ two band members, "`Meg`" and "`Jack`", each being paid a wage of £100 per week.
d. Finally, write the annual report for the White Stripes.

3. Managing bands is difficult; you need more control (and power!). Let's extend the capability of the Band class.

a. Edit the "`Band`" class to add a new method called "`getMembers`" that returns a python list of band members.
b. Add a method called "`sack`" that takes a band member as its argument and removes that member from the "`self.wages`" dictionary.
c. Add a method called "`promote`" that takes two arguments: a band `member` and the `payrise` you wish to give.
d. Start an interactive Python session and import the `Band` class from "`band.py`".
e. Create a variable "`hs`" as an instance of Band named "`Hearsay`".
f. Add the following band members (at £10 per week each): ("`Suzanne`", "`Danny`", "`Kym`", "`Myleene`", "`Noel`").
g. `Danny` has simply not been performing, use your "`hs`" instance to sack `Danny`.
h. Check that Danny has been removed by printing "`hs.getMembers()`".
i. Try to employ "`Madonna`" at £1000000 per week. Did it work?

Solution: Object Oriented Programming (OOP)

1.

```
Linux: $ cp example_code/band.py band.py
Linux: $ python -c "import band"
```

2.

```
Linux: $ python
>>> from band import Band
>>> ws = Band("The White Stripes")
>>> ws.employ("Meg", 100)
>>> ws.employ("Jack", 100)
>>> ws.writeAnnualReport()
Band name: The White Stripes

         Band member | Weekly Wage
                 Meg | 100.00
                Jack | 100.00
```

3.

```
Your new methods might look like:

    def getMembers(self):
        "Return a list of band members."
        members = self.wages.keys()
        return members

    def sack(self, member):
        "Removes a band member."
        del self.wages[member]

    def promote(self, member, payrise):
        "Increases wage of band member."
        self.wages[member] += payrise

Linux: $ python
>>> from band import Band
>>> hs = Band("Hearsay")
>>> applicants = ("Suzanne", "Danny", "Kym", "Myleene", "Noel")
>>> for app in applicants:
...     hs.employ(app, 10)
>>> hs.sack("Danny")
>>> print hs.getMembers()
['Myleene', 'Noel', 'Suzanne', 'Kym']
>>> hs.employ("Madonna", 100000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "band.py", line 28, in employ
  raise ValueError("{} costs too much - cannot join the band!".format(member))
ValueError: Madonna costs too much - cannot join the band!
```