Programming in Python

Basics

Sources and references

- Scientific Programming, Analysis and Visualization with Python
  - http://snowball.millersville.edu/~adecaria/ESCI386P/esci386_main.html
- Python: Essential Reference (4th ed.) by David M. Beazley
- Python Pocket Reference by Mark Lutz
- Much of the Python documentation, as well as that for its libraries, is online
  - Python documentation page: http://docs.python.org/genindex.html
  - MATPLOTLIB documentation page: http://matplotlib.sourceforge.net
  - Numerical Python (NumPy) documentation page: http://docs.scipy.org/doc/numpy/reference
  - ScientificPython (SciPy) documentation page: http://docs.scipy.org/doc/scipy/reference
- Google is your best friend: A web search for “How do I [whatever you want to do] in Python?” will often yield fruitful results

Introduction

1 **Main characteristics of Python**

2 **Python: syntax**

3 **Dynamic typing**

4 **Control structures**

5 **Functions**

6 **Data structures: string, list, tuple, disctionary**

7 **Examples**

Interpreted language

Python is an interpreted programming language

- The difference between an interpreted language and a compiled language (such as Fortran or C++) is
  - In a compiled language the entire source code text file is read and then and then converted into an executable code (in machine language) that can be directly executed by the operating system. This executable code is often optimized for the operating system
  - In an interpreted language the source code text file is read line-by-line and each statement converted into machine language and executed before the next line is read, or the text-file is read completely and converted into an intermediate code that is then further converted into machine language for execution by the operating system

Compiled vs interpreted language

- Compiled languages are generally more efficient and faster than interpreted languages, since the conversion to machine language only occurs one time, and then the machine language code can be executed whenever the program needs to be run. In an interpreted language, the interpretation process occurs every time the program is run
- Interpreted languages are often more flexible and changes can be easily made to the program at runtime

Dynamic typing

- Most variables do not need to be declared as real, integer, string, etc.
- The type of a variable is defined by the value assigned to it
- The type of a variable can change throughout the program execution
- The opposite of dynamically-typed is statically-typed. Fortran and C are examples of statically-typed languages.

Object oriented

- Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods
- Python is an object-oriented language, but we will often use it in a more traditional procedural or functional programming paradigm

Open source and free

- Unlike with MATLAB, there are no licenses required
- There are companies that put together high quality Python bundles and support them for a fee
- The advantage is that the libraries supported by these bundles work together with few bugs.
- However, the same libraries can be assembled and installed freely from other sources, though there may be some additional headaches sorting out dependencies, etc.

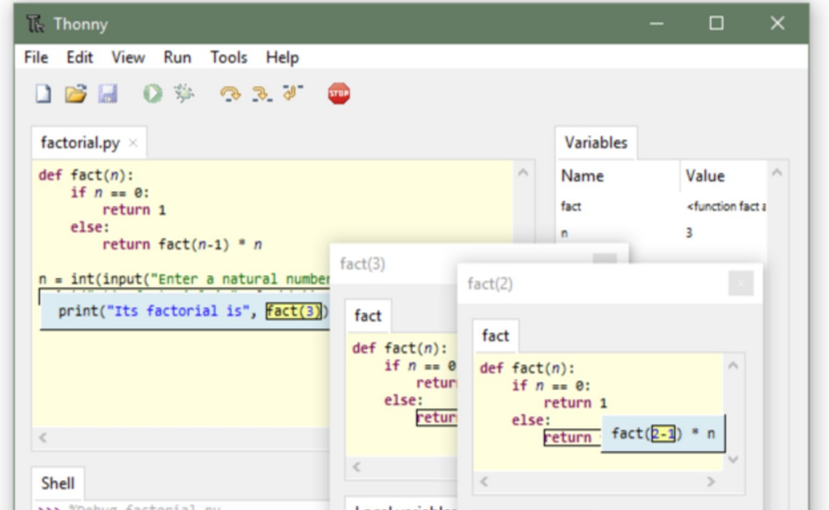
Indentation (code blocks)

- A code block consists of several lines of code that are uniformly indented
- Code blocks can be used with if, else, elif, for, and while statements, as well as others
- Most IDE allow you to select the number of spaces
- A good rule is using 4 spaces for indents, but you can use any number between 2 and 4
- Be uniform! Pick an indent number and stick with it

Thonny

Integrated Development Environment (IDE) specifically created for starting to program

+ is open source
+ is written in Python



Windows:
https://bitbucket.org/plas/thonny/downloads/thonny-2.1.22.exe
Mac:
https://bitbucket.org/plas/thonny/downloads/thonny-2.1.22.dmg
Linux:
https://bitbucket.org/plas/thonny/downloads/thonny-2.1.22.bash

Example

```
a = int(input("insert the number: "))
if (a == 5):
    print ("number equal to to 5")
else:
    print ("number different from 5")
print ("a = ", a)    # This instruction is executed anyway

>>> %Run first_example.py
insert the number5
number equal to to 5
a =  5
```

Continuation Line

- The \ character at the end of a line of Python code signifies that the next line is a continuation of the current line

Comments

- Comments in Python are indicated with a pound sign, #
- Any text following a # and before the next carriage return is ignored by the interpreter
- For multiple-line comments a # must be used at the beginning of each line

```
# Write a program that finds all the occurrences of the character ‘,’
# in a string and prints their position on screen.
# The string must be provided by the user
myString = input ("Give me the string with commas: ")
print ("You gave me the string: ", myString)
myList = []
for i in range (len(myString)):
    if (myString[i] == ','):
        print ("Comma found in position: ", i, "\n")
        myList.append(i)
print (myList)
```

`if` statement

- `if (cond)` executes a code block of instructions if `cond` is `True`, another block if `cond` is `False`

```
>>> a = 2
>>> if (a == 2):
...     print "executes if"
... else:
...     print "executes else"
...
executes if
>>> if (a == True):
...     print "executes if"
... else:
...     print "executes else"
...
executes else
```

Operators `and`, `or`, `not`

Operator	Value returned
`a or b`	**b**, if **a** is **False** or a value considered **False** **a** otherwise
`a and b`	**a**, if **a** is **False** or a value considered **False** **b** otherwise
`not a`	**True** if **a** is **False** or a value considered **False** **False** otherwise

- Decreasing precedence order: `not`, `and`, `or`

  `not a and b or c` equals to: `((not a) and b) or c`

- `not` has lower precedence with respect to other non-boolean operators

  `not a == b` equals to: `not (a==b)`

```
>>> True or False
True
>>> True and False
False
>>> not True
False
>>> not False
True
```

Comparison expressions

- A comparison expression returns a value of type `bool`
- Comparison operators

  `==` `<` `<=` `>=` `>` `!=` `is` `is not`
- Boolean values `False` and `True` are equals to integer 0 and 1 respectively
- Type conversion `int` → `float` are made if necessary when evaluating a comparison expression

```
>>> False == 0
True
>>> False == 1
False
>>> False == 2
False
>>> True == 0
False
>>> True == 1
True
>>> True == 2
False
>>> 1 == 2 - 1
True
>>> 1 == 1.0
True
>>> 1 == "1"
False
```

`if` statement (cont.)

```
a = int(input("Type a number and press ENTER: "))
print ("You typed ")
if (a > 0):
    print ("a positive number ( >0)")
elif (a < 0):
    print ("a negative number ( <0)")
else:
    print ("ZERO")
```

- parts `elif` and `else` are optional
- an arbitrary number of `elif` can be added
- `pass` statement produces no effect
- example of `pass`

```
if (a > 0):
    pass
b = 5
```

Exercise

Create a program that reads an input from the user and controls if such input is an integer and, in this case, prints if the integer is positive, negative, or zero

Exercise: first solution

```
# Create a program that reads an input from the user and
# controls if such input is an integer and, in this case,
# prints if the integer is positive, negative, or zero

a = int(input("Dammi il numero: "))
print("Il numero", a, "è intero")
if (a > 0):
    print("Il numero", a, "è positivo")
elif (a < 0):
    print("Il numero", a, "è negativo")
else:
    print("Il numero è 0")
```

Exceptions

- Errors detected during execution are called exceptions
- Most exceptions are not handled by programs, and result in error messages

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

- Exceptions come in different types, and the type is printed as part of the message, e.g. ZeroDivisionError, ValueError

Handling exceptions

Example:

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops!  That was no valid number.  Try again...")
```

```
Please enter a number: a
Oops!  That was no valid number.  Try again...
Please enter a number: asdas
Oops!  That was no valid number.  Try again...
Please enter a number: 1.2
Oops!  That was no valid number.  Try again...
Please enter a number: 1
>>>
```

Exercise: (clean) solution

```
# Create a program that reads an input from the user and
# controls if such input is an integer and, in this case,
# prints if the integer is positive, negative, or zero

a = input("Dammi il numero: ")
try:
    a = int(a)
except ValueError:
    print("Non è int")
print("Il numero", a, "è intero")
if (a > 0):
    print("Il numero", a, "è positivo")
elif (a < 0):
    print("Il numero", a, "è negativo")
else:
    print("Il numero è 0")
```

Loops

- Looping in Python is accomplished using either the for or the while statements
- A common way to loop is to use the for statement
- However, in python, the for statement requires an iterable object such as a list, a tuple, a range, an array, or even a string
- while statement is simpler

`while` loop

- `while` loops while a condition is True

```
a = int(input("Type a number and press ENTER (0 to terminate): "))
while (a != 0):
    print ("You typed: ", a)
    a = int(input("Type a number and press ENTER (0 to terminate):"))
print ("You typed 0 and program terminates.")
```

- `break` statement breaks the loop
- `continue` statement jumps to next iteration

```
count = 0
while (True):
    a = int(input("Type a number and press ENTER (0 to terminate): "))
    if (a > 0):
        print ("You typed a positive number: it will be counted")
    elif (a < 0):
        print ("You typed a negative number: it will NOT be counted")
        continue
    else:
        print ("You typed 0 and program terminates.")
        break
    count = count + 1

print ("You typed %d positive numbers." % count)
```

While exercise

Create a program that reads an input from the user, writes the value provided by user on screen, and loops for maximum 10 times or until the user types a 0

While exercise: solution

```
# Create a program that reads an input from the user,
# writes the value provided by user on screen,
# and loops for maximum 10 times or until the user types a 0
loops = 0
while (loops < 10):
    a = int(input("Type a number and press ENTER (0 to terminate): "))
    print ("You typed %d" % a)
    if (a == 0):
        print ("Program terminates.")
        break
    else:
        loops = loops + 1
        continue
print ("You made %d loops." % loops)
```

`for` loop

- **`for`** loop executes a block for all the elements of an iterable data structure

```
for iterating_var in sequence:
    corpo_del_ciclo
```

- For each pass through the loop the next item in the iterable object is passed to the iterating_var
- iterating_var can be any valid variable name
- It should be a new variable, not one already used

```
names = ['Antonio', 'Mario', 'Giuseppe', 'Francesco']
for name in names:
    print ('Name :', name)
```

`for` loop

- To iterate on an integer index, there is the range() function

```
for n in range(-5,30,5):
    print(n)
```

- It is also possible to iterate on a list through an integer index

```
names = ['Antonio', 'Mario', 'Giuseppe', 'Francesco']
for index in range (len(names)):
    print ('Name :', names[index])
```

- **`break`** nel corpo del ciclo fa uscire dal ciclo
- **`continue`** nel corpo del ciclo salta all’iterazione successiva

Data structures

- One of the peculiar aspects of the Python language is the wealth of data structures defined by the language
- In particular, the language offers the programmer various data structures suitable to contain objects of various types
  - lists, tuples, dictionaries, strings, set
  - the collections module defines additional container types
- The various data structures differ in various aspects
  - the way in which the objects contained can be accessed
  - the possibility of iterating over the elements contained in the data structure
  - the possible order defined among the elements contained
  - the possibility or not to modify the elements present in the data structure once it has been "constructed"
    - we speak of mutable or immutable data structures
    - lists, dictionaries, sets are mutable while strings and tuples are immutable

string

- In Python a string of characters can be defined through one, two, or three quotes

  single quote ’exampleString’ double quote "exampleString"

- Mixing single, double, and triple quotes allows quotes to appear within strings

```
>>> s = 'Dad said, "Do it now!"'
>>> s
'Dad said, "Do it now!"`
>>> print(s)
Dad said, "Do it now!"
```

- Triple-quoted strings can include multiple lines, and retain all formatting contained within the triple quotes

```
>>> s = '''This sentence runs
over a
few lines.'''
>>> s
'This sentence runs\n over a\n few lines.'
>>> print(s)
This sentence runs
over a
few lines.
```

string (cont.)

- Special characters within string literals are preceded by the backslash, \
- One common special character is the newline command, \n, which forces any subsequent text to be printed on a new line
- Strings are formatted in Python using the .format() method

```
>>> x = 654.589342982
>>> s = 'The value of x is {0:7.2f}'.format(x)
>>> s
'The value of x is 654.59'
```

- Strings can be cat using + operator
- Single characters of a string are identified through an integer index
    - s[0] is the first character, s[1] the second one, etc.
- Negative index values can also be used to start from the end
    - s[-1] is the last, s[-2] is the second-last, etc.

string: examples

```
>>> print ('row 1\nrow 2')
row 1
row 2
>>> name = "Alessio "
>>> print ("Hello " + name)
Hello Alessio
>>> print ("Hello"*3)
HelloHelloHello
>>> s = '''This is a string
... that contains
... three lines'''
>>> print (s)
This is a string
that contains
three lines
>>> s1 = "ABC"; print (s1[0])
A
>>> s1[0] = 'X'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: ’str’ object does not support
item assignment
```

```
>>> s = "ABCDEFGHIJ"
>>> print (s[0]); print (s[1])
A
B
>>> print (s[9])
J
>>> print (s[10])
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> print (s[-1]); print (s[-2])
J
I
>>> print (s[-10])
A
```

String slicing

- It is possible to construct a new string starting from pieces of other strings
- The index notation can be used, specifying the index of the first (comprised) character and that of the last (excluded) separated by a colon

```
s[i:j]
```

- The substring starts from the first character if the first index is omitted
  - `s[:i]`
- The substring ends with the last character if the second index is omitted
  - `s[i:]`

String slicing examples

```
>>> s = "Qui, Quo, Qua"
>>> s[0:3]
'Qui'
>>> s[:3]
'Qui'
>>> s[3:3]
''
>>> s[3:4]
','
>>> s[5:8]
'Quo'
>>> s[ -3: -1]
'Qu'
>>> s[ -3:]
'Qua'
>>>
```

String main methods (1)

A string is an *object* with several predefined methods

- **`s.find(sub_str)`** returns the index of the position of the first occurrence of the substring **`sub_str`** in the string **`s`**
- **`s.find(sub_str, start)`** returns the index of the position of the first occurrence of the substring **`sub_str`** in the string **`s`**, starting from index **`start`**
- **`find()`** method returns -1 if the substring **`sub_str`** has not been found
- **`s.split(sep)`** returns a list of substrings of **`s`** separated by separator character **`sep`**. If **`sep`** does not exist in **`s`**, **`split()`** returns the list with the single element **`s`**

```
>>> s = "Qui, Quo, Qua"
>>> s.find(", ")
3
>>> s.find(", ", 0)
3
>>> s.find(", ", 3)
3
>>> s.find(", ", 4)
8
>>> s.find("X")
-1
>>> s.split(",")
['Qui', ' Quo', ' Qua']
>>> s.split(", ")
['Qui', 'Quo', 'Qua']
>>> s.split("X")
['Qui, Quo, Qua']
```

String main methods (2)

- `s.strip()` returns a string obtained deleting from `s` characters space, tab (\t), newline (\n) located at left and right ends
- `s.rstrip()` returns a string obtained deleting from `s` characters space, tab (\t), newline (\n) located at right ends
- `s.lstrip()` returns a string obtained deleting from `s` characters space, tab (\t), newline (\n) located at left ends
- `s.startswith(x)` returns `True` if the string `s` starts with the substring `x`, `False` otherwise
- `s.endswith(x)` returns `True` if the string `s` ends with the substring `x`, `False` otherwise
- `s.upper(x)` returns a string with the characters of `s` that are in lower case put in upper case, the other characters remain unchanged
- `s.lower(x)` returns a string with the characters of `s` that are in upper case put in lower case, the other characters remain unchanged

```
>>> s = "  Qui,\tQuo,\tQua\t\n"
>>> print s
  Qui,    Quo,    Qua
>>> s.strip()
'Qui,\tQuo,\tQua'
>>> s.rstrip()
'  Qui,\tQuo,\tQua'
>>> s.lstrip()
'Qui,\tQuo,\tQua\t\n'
```

```
>>> s = "Qui, Quo, Qua"
>>> s.startswith('Qui')
True
>>> s.endswith('Qua')
True
>>> s.upper()
'QUI, QUO, QUA'
>>> s.lower()
'qui, quo, qua'
>>>
```

String exercise 1

Write a program that finds all the occurrences of the character ‘,’ in a string provided by the user and prints on screen their position.

String exercise 2

Write the same program using find() method and while() loop.

List

- Lists are data structures ordered and mutable that can contain objects of different type
- To create a list associated to name **l**

```
l = [1, 2, "Pippo"]
```

- The simbols **[]** indicate an empty list
- operators + and * work on lists as done on strings
- Function **len(l)** returns the number of elements in **l**
- Such elements can be identified through an index, as for strings (and arrays in C/C++)
- A sublist containing all the elements of **l** comprised between that of index **i** (comprised) and that of index **j** (excluded) can be identified through **l[i:j]**
  - If **i** is not specified, it means **i=0**
  - If **j** is not specified, it means **j=len(l)**

```
>>> l = [1, 2]
>>> m = ["Pippo", 3, 4]
>>> print (l + m)
[1, 2, 'Pippo', 3, 4]
>>> print (l*4)
[1, 2, 1, 2, 1, 2, 1, 2]
>>> print (m[0])
Pippo
>>> m[0] = 99; print (m)
[99, 3, 4]
>>> len(m)
3
```

```
>>> n = l + m [1:]; print (n)
[1, 2, 3, 4]
>>> print (n [1:2])
[2]
>>> print (n [1:3])
[2, 3]
>>> print (n [1:4])
[2, 3, 4]
>>> print (n [1:5])
[2, 3, 4]
>>> print (n [5:6])
[]
```

List main metods

- **`l.append(obj)`** appends **`obj`** at the tail of list **`l`**
- **`l.extend(l1)`** extends **`l`** adding the elements of list **`l1`** at the tail of list **`l`**
- **`l.insert(index, obj)`** adds **`obj`** in **`l`** before the position indicated by **`index`**
- **`l.pop(index)`** removes from **`l`** the object in position **`index`** and returns it
- **`l.remove(obj)`** removes the first occurrence of **`obj`** from list **`l`**
- **`l.reverse()`** inverts the order of the elements **`l`**
- **`l.sort()`** sorts the elements of **`l`** in increasing order
- **`l.index(obj)`** returns the index of the first occurrence of **`obj`** in list **`l`**
- **`l.count(obj)`** returns the number of occurrence of **`obj`** in list **`l`**

```
>>> l1 = [6, 7, 8, 9]
>>> l2 = [1, 2, 3, 4, 5]
>>> l1.append(10); print (l1)
[6, 7, 8, 9, 10]
>>> l1.extend(l2); print (l1)
[6, 7, 8, 9, 10, 1, 2, 3, 4, 5]
>>> l1.insert(0, "START"); print (l1)
['START', 6, 7, 8, 9, 10, 1, 2, 3, 4, 5]
>>> l1.insert(6, "x"); print (l1)
['START', 6, 7, 8, 9, 10, 'x', 1, 2, 3, 4, 5]
>>> l1.pop(0); print (l1)
'START'
[6, 7, 8, 9, 10, 'x', 1, 2, 3, 4, 5]
>>> l1.remove("x"); print (l1)
[6, 7, 8, 9, 10, 1, 2, 3, 4, 5]
```

```
>>> l1.reverse(); print (l1)
[5, 4, 3, 2, 1, 10, 9, 8, 7, 6]
>>> l1.sort(); print (l1)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> l1.pop( -1)
10
>>> l1.index(1)
0
>>> l1.index(9)
8
>>> l1.index(10)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: 10 is not in list
```

Lists: other useful functions

- **`range(n)`** returns a list composed on integers comprised between 0 and n-1
- **`range(m, n, step)`** returns a list composed of integers comprised between m (included) and n (excluded), with an increment of step
- **`len(x)`** returns the number of elements of list x
- **`max(l)`** returns the maximum element of a list

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(1, 6, 2)
[1, 3, 5]
>>> len([0, 1, 2, 3])
4
>>> max([1.0, 2.5, -2.3, 1.3])
2.5
```

List exercise 1

Design and develop a program that creates a list with values provided by the user, then asks the user for a certain value, finds this value in the list and shows the value of the item in the list preceding the found one.

List exercise 1: solution

```
element_list = []
maxlen = 5

# Insert elements
while len(element_list) < maxlen:
    element = input("Enter your element:")
    element_list.append(element)
    print(element_list)

# Find an element
find_element=input("Enter an element you want to search for in the list:")
for i in range( len( element_list ) ):
    if ( element_list[ i ] == find_element ):
        previous_index = i - 1
print("The previous element in the list is:", element_list[previous_index])
```

Tuple

- Tuples are *immutable* data structures that can contain objects of different type
    - `t = (1, 2, "Pippo")`
- `()` is an empyt tuple, `(a,)` is a tuple with the only element `a`
- Operators + and * also work on tuples
- Function `len(t)` returns the number of elements in a tuple `t`
- `t[i:j]` is tuple made of elements of `t` comprised between that of index `i` (comprised) and that of index `j` (excluded)
    - If `i` is not specified, it means `i=0`
    - If `j` is not specified, it means `j=len(t)`

```
>>> t1 = (1, 2)
>>> t2 = ("Pippo", 3, 4)
>>> t = t1 + t2; print (t)
(1, 2, 'Pippo', 3, 4)
>>> len(t)
5
>>> print (t[0])
1
>>> print (t[2])
Pippo
```

```
>>> t[0] = 99
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not
support
item assignment
>>> print (t[1:3])
(2, 'Pippo')
>>> print (t[1:1])
()
>>> print (t[5:5])
()
```

Dictionaries

- A dictionary is a structure that contains couples (key, values), where every value is identified by a key
- This is an dictionary called **`d`**:

```
d = {key1: val1, key2: val2, key3: val3}
```

- Access to elements of d is made through the key

```
d[key]
```

- If the value of the key does not exist, an error is returned
- Dictionaries are mutable structures, while the value of the key is not mutable
- The operator **`in`** returns **`True`** if the key is in the dictionary

```
>>> d = { "NA": "Napoli",    "AV": "Avellino",
          "BN": "Benevento", "CE": "Caserta",
          "SA": "Salerno"}
>>> d["BN"]
'Benevento'
>>> d["MI"]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'MI'
>>> d[0]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 0
```

```
>>> d["NA"] = "Naples"
>>> print (d["NA"])
Naples
>>> "NA" in d
True
>>> "AV" in d
True
>>> "MI" in d
False
```

Dictionary main methods

- `d.clear()` deletes all the elements of dictionary `d`
- `d.copy()` returns a copy of dictionary `d`
- `d.has_key(key)` returns `True` if `key` exists in `d`
- `d.items()` returns a list with all the couples (key, value) in `d`
- `d.keys()` returns a list with all the keys in `d`
- `d.values()` returns a list with all the values in `d`
- `d.update(d2)` adds the content of `d2` to that of `d`
- `d.get(key,val)` returns the value associated with `key`, `val` otherwise
- `d.get(key)` returns the value associated with `key`, `None` otherwise

```
>>> d = {"NA":"Napoli","AV":"Avellino","BN":"Benevento","CE":"Caserta","SA":"Salerno"}
>>> d.has_key("NA")
True
>>> d.has_key("MI")
False
>>> d.items()
[('NA', 'Napoli'), ('BN', 'Benevento'), ('SA', 'Salerno'), ('CE', 'Caserta'), ('AV', 'Avellino')]
>>> d.keys()
['NA', 'BN', 'SA', 'CE', 'AV']
>>> d.values()
['Napoli', 'Benevento', 'Salerno', 'Caserta', 'Avellino']
>>> print d.get("NA")
Napoli
>>> print d.get("MI")
None
>>> d.update({"NA": "Naples", "MI": "Milano"})
>>> print d
{'AV': 'Avellino', 'NA': 'Naples', 'BN': 'Benevento', 'MI': 'Milano', 'SA': 'Salerno', 'CE': 'Caserta'}
```

Dictionary exercise 1

Design and implement a program that manages a price list. The program should ask user to add articles to the price list. For each article, the program asks the name and price, stores this article in the data structure and then asks the user if he/she wants to add more articles. At the end of this phase, the program prints the list price on screen.