Programming Python

Derek Karssenberg

Book

Lecture and exercises are based on the book:
Think Python, How to Think Like a Computer Scientist

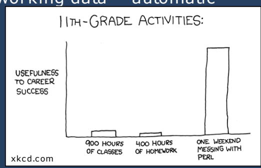
Freely available via:
http://www.greenteapress.com/thinkpython/

Topics

- Choosing a programming language
- Python applications
- Variables, expressions and statements
- Functions
- Conditionals and user intervention
- Fruitful functions and program development
- Strings
- Lists
- Files and exceptions

Why learn programming?

Structuring your work
- Repeatable and fast
- Separate source data and 'working data' - automatic conversion by a program!



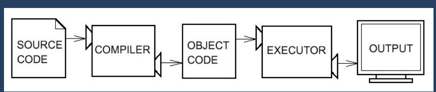
Developing models
- Combined with PCRaster

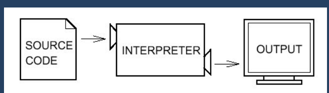
or other modules

Other reasons
- Other software, e.g. developing a www site, creating a graphical user interface

Choosing a language (1)

Compiled versus interpreted programming languages:





Choosing a language (2)

Low-level languages versus high-level languages
- Low-level language: concepts of computer language is similar to concepts of a computer
- High-level language: concept of computer language is closer to how humans think

Low-level language: example program (C++)

To print

Hello, world.

on the screen, we need in C++ the following program

```
#include <iostream.h>

void main()
{
  cout << "Hello, world." << endl;
}
```

High-level language: example program (Python)

To print

Hello, world.
on the screen, we need in Python the following program

```
print "Hello, world."
```

Choosing a language (3)

Compared to low-level languages, a high-level language
- results in shorter programs
- is easier to learn
- results in longer runtimes (but not always)

Examples of computer languages
- Machine languages: compiled, low-level
- C++, Fortran, Java: compiled, low-level
- Perl, Python, PCRaster, MATLAB: interpreted, high-level

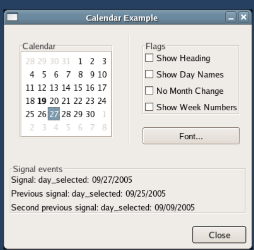
Why Python?

- High-level language: easier to learn
- Free and open source software
- Runs on all platforms (i.e. Microsoft Windows, Linux, Unix, Apple Macintosh)
- Comes with many modules (preprogrammed stuff)
- Common in the GIS world
- Used as framework for spatio-temporal modelling in PCRaster

Website and software: http://www.python.org

Example Python applications (1)

- Widgets

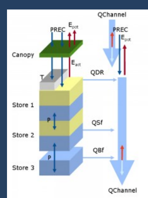


Example Python applications (2)

- Simulation models

Like spatio-temporal modelling with PCRaster

http://pcraster.geo.uu.nl/projects/applications/pcrglobwb/



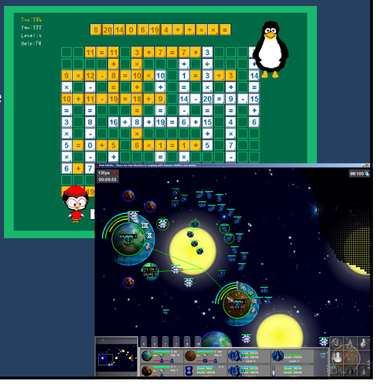
Model concept of PCR-GLOBWB.

Example Python applications (3)

- Games

Like Tux Math Scrabble or Void Infinity

www.pygame.org



Variables, expressions and statements

Types

Values have a type: string, integer, floating-point or Boolean

String
"This is a string", or "0.234", or " " (whitespace)

Used for:

- proper names
- text printed on the screen or written to a file

Types

Values have a type: string, integer, floating-point or Boolean

Integer
2, or 3, or -2, or 0, not 0.0!

Used for:

- Classes, e.g. id's of provinces
- counters (e.g., 0,1,2,3,4...100)

Types

Values have a type: string, integer, floating-point or Boolean

Floating-point
2.234, or -12.3234, or 2343.1, or 0.0

Used for:

- scalar values used in calculations, e.g. elevation

Types

Values have a type: string, integer, floating-point or Boolean

Boolean
0 (FALSE) or 1 (TRUE)

Used for:

- result of comparisons
- conditions

Variables (1)

A variable is a way to reference to a known or unknown value

Assigning a value to a variable:
- streamPower = 23.4
- myName = "Piet‟

Variables (2)

Meaning of “=‟ is
- equality in mathematics
- assignment in Python, assigning a value to a variable

Equality in Python is “==‟
This will be discussed later.

Variables (3)

Some rules:
- use meaningful names
- no spaces and preferably no underscores
- first letter a lowercase

e.g.,

```
streamPower
```

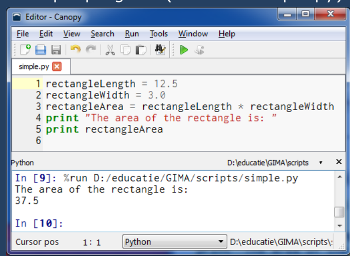
instead of:

```
StreamPower
stream Power
stream_power
```

Expressions

An expression is an instruction to execute something

A simple program (saved as simple.py):



Python command line mode

At the prompt, type:

```
python <Enter>
```

And you get the python prompt:

```
>>>
```

Enter single statements, e.g.:

```
>>> 2*3
6
>>> a = 2.5
>>> b = 3
>>> c = a * b
>>> c
7.5
```

Creating and running a Python program/script

A python program is an ascii file
- Edit with any ascii editor (e.g. edit, vi, Wordpad etc)
- Or use editors specifically for Python (e.g. IDLE, Canopy, Spyder)

Executing a python program
- type on the command line:

```
python myProgram.py
```

- or use the ‘Run’ button in a dedicated editor

All statements will be executed from top to bottom!

Functions

Operators, syntax

Syntax:
rV = *arg1* operatorName *arg2*

with:
- *rV*: return value
- *arg1, arg2*: arguments
- operatorName: name of the operator

The operator 'reads' the inputs (arguments), does 'something' and assigns values to its outputs, the arguments.

Example:
a = b * c

Functions, syntax

Syntax:
rV1, rV2,..,rVn = functionName(*arg1, arg2,..,argm*)

with:
- *rV1, rV2,..,rVn*: return values 1..*n*
- *arg1, arg2,..,argm*: arguments 1..*m*
- functionName: name of the function

The function 'reads' the inputs (arguments), does 'something' and assigns values to its outputs, the arguments.

Using functions, example (1)

The function float reads the value of the argument, converts it to a floating-point and returns a floating-point value:

```
# making a float
anInteger=2
aFloatingPoint=float(anInteger)
```

A hashtag (#) makes that the expression after it is not executed. Can be used to:
- put comments in the script (do this!)
- (temporarily) comment out parts of the script, e.g. when testing

Using functions, example (2)

The function string.capitalize returns a copy of its input argument (a string), with the first character capitalized:

```
import string

aName="piet"
aNameCapitals=string.capitalize(aName)
print aNameCapitals
```

When executing this script (name.py), it prints:

```
Piet
```

Using functions, example (3)

The function string.replace returns a copy of its input argument (a string), with a part of the string replaced with another string:

```
import string

aName="piet"
aNewName=string.replace(aName,"iet",
"eter")
print aNewName
```

When executing this script (name2.py), it prints:

```
peter
```

Modules/libraries

A module is a file with a collection of related functions. It needs to be imported at the top of a program, e.g.:

```
import string
import math
```

Functions from a module are called using dot notation, e.g.:

```
aNewName=string.replace(aName,
"iet", "eter")
logRunoff=math.log10(runoff)
```

Creating functions

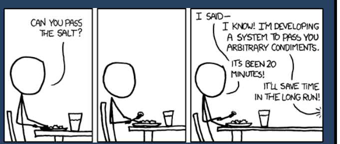
Python comes with many built-in functions (most of them in modules)

You can also create functions yourself

- new functions are built as a combination of existing python components (expressions)
- the definition of a new function is given in the main program or in an associated file
- a new function can be used anywhere in the program

Why creating functions?

- To group statements serving one purpose; this makes the program easier to read and to debug
- To make the script shorter by eliminating repetitive code
- If you want to change something in the function you only have to do it once, in the repetitive code this would be several times
- Functions can be reused by others or in other programs of your own



Function definition, syntax

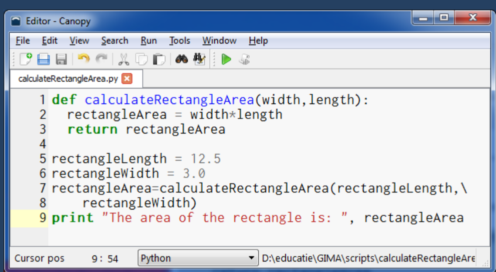
def *functionName*(*arg1,arg2,..,argn*):
    statement1
    ..
    statementm
    return *varReturn1,...,varReturnl*

with:

- *functionName*: the name of the new function
- *arg1, arg2, ..., argn*: input arguments
- *statement1, ...,statementm*: statements doing something with the inputs
- *varReturn1, ...,varReturnl*: variables returned by the function

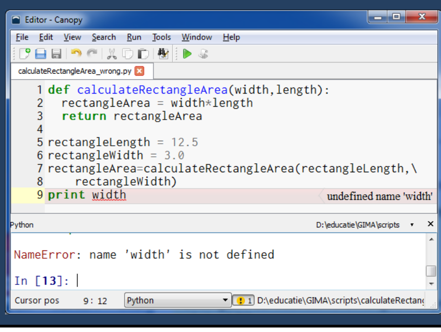
Function definition, example

The function calculateRectangleArea with two input arguments returns one value:



Function definition, example

A variable created in a function does not exist outside the function! E.g.:



Conditionals and user intervention (and comparison operators, Boolean operators)

Comparison operators

Comparison operators compare two values or, more commonly, variables

```
x == y       # TRUE if x is equal to y
x != y       # TRUE if x is not equal to y
x > y        # TRUE if x is greater than y
x < y        # TRUE if x is less than y
x >= y       # TRUE if x is greater than or equal to y
x <= y       # TRUE if x is less than or equal to y
```

The result of comparison operators is a 0 (FALSE) or 1 (TRUE), of type Boolean.

Comparison operators

The result of comparison operators is a 0 (FALSE) or 1 (TRUE), of type Boolean.

```
a = 4>3
print a
print(type(a))
```

```
1
<type 'bool'>
```

Logical (Boolean) operators

Evaluate the logical relation between two values or variables

```
x and y      # TRUE if both x and y are TRUE
x or y       # TRUE if x or y are TRUE
not x        # TRUE if x is FALSE
```

The operands (x and y above) are in most cases Booleans where:

- a 0 is considered FALSE
- a value unequal to 0 is considered TRUE

The result of logical operators is a 0 (FALSE) or 1 (TRUE), of type Boolean.

Combining comparison and logical operators

For instance:

```
(a >= b) and not (d < c)

(2*a < 100.0) or (b/3 > c)
```

Conditional statements, syntax

A conditional statement checks whether a condition is fulfilled and only if it is, it executes a block of code:

```
if CONDITION:
  STATEMENT1
  ...
  STATEMENTn
```

with:

- CONDITION, an expression with a Boolean result
- STATEMENT1,..,STATEMENTn, statements which are executed if the CONDITION is TRUE

Conditional statements, example (1)

```
if (rain > 0):
  print "stay at home!"
```

Conditional statements, example (2)

```
if (x >= 0):
  sqrtX=math.sqrt(x)
  print "the square root of x is ", x
```

Conditional statements and alternatives, syntax

You can also define a block of code that is executed if the condition is **not** fulfilled:

```
if CONDITION:
 STATEMENT1
 …
 STATEMENTn
else:
 ALTSTAT1
 ..
 ALTSTATm
```

- ALTSTAT1..ALTSTATm, statements which are executed if the CONDITION is FALSE

Conditional statements, example (1a)

```
if (rain > 0):
  print "stay at home!"
else:
  print "go swimming!"
```

Conditional statements, example (2a)

```
if (x >= 0):
  sqrtX=math.sqrt(x)
  print "the square root of x is ", x
else:
  print "the square root cannot be calculated since\
      x is negative!"
```

Conditional statements chained, syntax

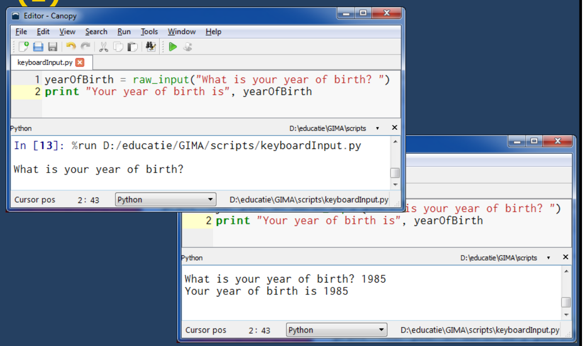
You can also chain different conditional statements. The second is checked if the first is not fulfilled:

```
if CONDITION:
 STATEMENT1
 ...
 STATEMENTn
elif ANOTHERCOND:
 ALTSTAT1
 ..
 ALTSTATm
else:
 ALTALTSTAT1
 ..
 ALTALTSTATl
```

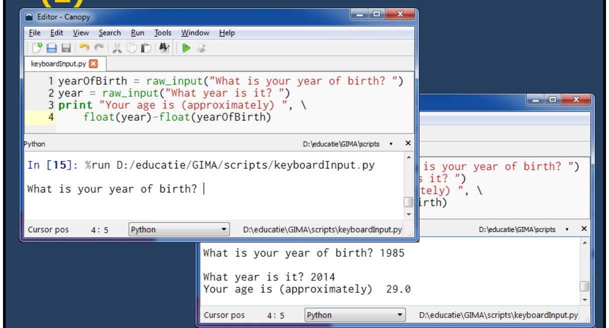
Conditional statements, example (1b)

```
if (rain > 0):
  print "stay at home!"
elif (temperature > 30):
  print "go swimming!"
else:
  print "have a drink on a terrace!"
```

User intervention: keyboard input (1)



User intervention: keyboard input (2)



Fruitful functions and program development

- Loops
- Encapsulation
- Generalization
- Local variables

Loops, the for statement, syntax

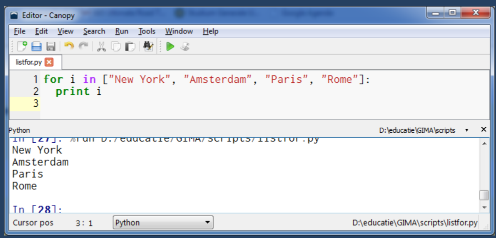
The for statement is used for loops when you already know in advance how many iterations are needed.

```
for ELEMENT in COMPOUND:
  STATEMENT1
  …
  STATEMENTn
```

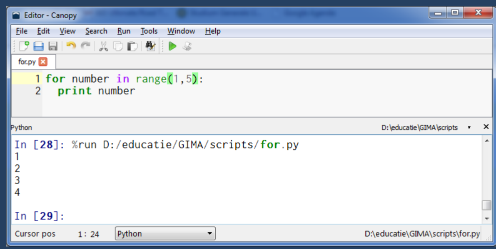
with

- ELEMENT, an element which can be of any type
- COMPOUND, a compound data type, e.g. a list (explained later)
- STATEMENT1,..,STATEMENTn, the statements in the 'body' of the while statement

For statement, example (1)



For statement, example (2)



Loops, the while statement, syntax

The while statements is used for loops when you do **not** know how many iterations are needed.

```
while CONDITION:
  STATEMENT1
  …
  STATEMENTn
```

with

- CONDITION, a Boolean expression
- STATEMENT1,..,STATEMENTn, the statements in the 'body' of the while statement
- Note: STATEMENT1,..,STATEMENTn generally determine CONDITION

Loops, the while statement, example (1)

```
# program with a while loop
n = 0
while n < 20:
  print n,
  n = n+1
```

Operation:

- evaluate CONDITION, yielding TRUE or FALSE
- if CONDITION is FALSE, exit the while statement, and continue the program below the while statement
- if CONDITION is TRUE, execute STATEMENT1,..,STATEMENTn, and go back to step 1

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

Loops, the while statement, example (2)

```
# program with a while loop
n = 0
while n < 20:
  print n,
  n = n+1
print "The value of n after the loop is:", n
```

Question: What does this print?

Loops, the while statement, example (2)

```
# program with a while loop
n = 0
while n < 20:
  print n,
  n = n+1
print "The value of n after the loop is:", n
```

Question: What does this print?

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
The value after the loop is: 20
```

Loops, the while statement, example (3)

```
# program with a while loop
n = 0
while 1:
  print n,
  n = n+1
print "The value of n after the loop is:", n
```

Question: What does this print?

Loops, the while statement, example (4)

```
# program with a while loop
n = 0
while n < 20:
  print n,
  n = n+1
print "The value of n after the loop is:", n
```

Change into:

```
# program with a while loop
n = 40
while n < 20:
  print n,
  n = n+1
print "The value of n after the loop is:", n
```

```
The value after the loop is: 40
```

While statement, printing a table

```
import math

print 'degrees\tfraction (m/m)'
slopeDegrees = 0.0
while slopeDegrees < 30.0:
  slopeRadians=(slopeDegrees/360)*2*math.pi
  slopeFraction=math.tan(slopeRadians)
  print slopeDegrees, '\t', slopeFraction
  slopeDegrees=slopeDegrees+5.0
```

prints:

degrees	fraction (m/m)
0.0	0.0
5.0	0.0874886635259
10.0	0.176326980708
15.0	0.267949192431
20.0	0.363970234266
25.0	0.466307658155

Creating functions (1)

Rewrite the code in the previous slide as (encapsulation):

```
import math

def degreesToRadians(degrees):
  radians=(degrees/360)*2*math.pi
  return radians


print 'degrees\tfraction (m/m)'
slopeDegrees = 0.0
while slopeDegrees < 30.0:
  slopeRadians=degreesToRadians(slopeDegrees)
  slopeFraction=math.tan(slopeRadians)
  print slopeDegrees, '\t', slopeFraction
  slopeDegrees=slopeDegrees+5.0
```

Creating functions (2)

Or even as:

```
import math

def degreesToRadians(degrees):
  radians=(degrees/360)*2*math.pi
  return radians

def degreesToFraction(degrees):
  radians=degreesToRadians(degrees)
  fraction=math.tan(radians)
  return fraction

print 'degrees\tfraction (m/m)'
slopeDegrees = 0.0
while slopeDegrees < 30.0:
  slopeFraction=degreesToFraction(slopeDegrees)
  print slopeDegrees, '\t', slopeFraction
  slopeDegrees=slopeDegrees+5.0
```

Creating functions (3)

```
import math

def degreesToRadians(degrees):
  radians=(degrees/360)*2*math.pi
  return radians

def degreesToFraction(degrees):
  radians=degreesToRadians(degrees)
  fraction=math.tan(radians)
  return fraction

def printDegreesToFractionTable():
  print 'degrees\tfraction (m/m)'
  slopeDegrees = 0.0
  while slopeDegrees < 30.0:
    slopeFraction=degreesToFraction(slopeDegrees)
    print slopeDegrees, '\t', slopeFraction
    slopeDegrees=slopeDegrees+5.0

printDegreesToFractionTable()
```

Why is encapsulation useful?

- Program is easier to read
- Reuse of code
- Easy debugging

```python
import math

def degreesToRadians(degrees):
  radians=(degrees/360)*2*math.pi
  return radians

aRadian = degreesToRadians(30)
```

Generalization

```python
import math

def degreesToRadians(degrees):
  radians=(degrees/360)*2*math.pi
  return radians

def degreesToFraction(degrees):
  radians=degreesToRadians(degrees)
  fraction=math.tan(radians)
  return fraction

def printDegreesToFractionTable():
  print 'degrees\tfraction (m/m)'
  slopeDegrees = 0.0
  while slopeDegrees < 30.0:
    slopeFraction=degreesToFraction(slopeDegrees)
    print slopeDegrees, '\t', slopeFraction
    slopeDegrees=slopeDegrees+5.0

printDegreesToFractionTable()
```

Turn this script into:

Generalization (2)

```python
import math

def degreesToRadians(degrees):
  radians=(degrees/360)*2*math.pi
  return radians

def degreesToFraction(degrees):
  radians=degreesToRadians(degrees)
  fraction=math.tan(radians)
  return fraction

def printDegreesToFractionTable(min,max,step):
  print 'degrees\tfraction (m/m)'
  slopeDegrees = min
  while slopeDegrees < max:
    slopeFraction=degreesToFraction(slopeDegrees)
    print slopeDegrees, '\t', slopeFraction
    slopeDegrees=slopeDegrees+step

printDegreesToFractionTable(0,30,5.0)
```

Generalization (3)

```
printDegreesToFractionTable(10,30,1.0)
```

will print this table:

```
10      0.0
11.0    0.194380309138
12.0    0.21255656167
13.0    0.230868191126
14.0    0.249328002843
15.0    0.267949192431
16.0    0.286745385759
17.0    0.305730681459
18.0    0.324919696233
19.0    0.34432761329
20.0    0.363970234266
21.0    0.383864035035
22.0    0.404026225835
23.0    0.42447481621
24.0    0.445228685309
25.0    0.466307658155
26.0    0.487732588566
27.0    0.509525449494
28.0    0.531709431661
29.0    0.554309051453
```

Local variables (1)

Variables created in a <u>function</u> are local variables:
→ they are not known outside the function

E.g. this program:

```python
def aFunction():
  n = 0

aFunction()
print n
```

```
Traceback (most recent call last):
  File "local0.py", line 5, in ?
    print n
NameError: name 'n' is not defined
```

Local variables (2)

Variables created in a <u>function</u> are local variables:
→ they are not known outside the function
→ they do not affect variables outside the function

```python
def aFunction():
  n = 0
  print "n inside the function:", n

n = 100
aFunction()
print "n outside the function:", n
```

```
n inside the function: 0
n outside the function: 100
```

Local variables (3)

Also, variables in a loop are NOT local variables:

E.g. this program:

```
n = 0
while n < 10:
  n = n+1

print n
```

```
10
```

Strings

Compound data type, syntax of bracket operator

Compound data type: data type consisting of smaller pieces

Data type string: compound data type consisting of letters

Selecting a single string with the bracket [] operator:

```
LETTER = STRING[J]
```

with:

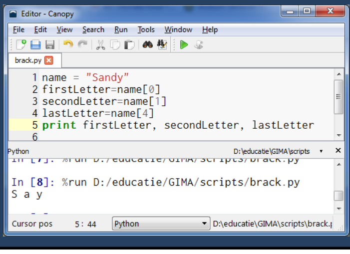
- STRING, a variable of data type string
- J, index, a variable of data type integer
- LETTER, a letter of STRING (note: LETTER is also of type string)

Bracket operator, non-negative index

```
LETTER = STRING[J]
```

If J ≥ 0:
LETTER is the (J+1)-eth letter of STRING
So the first element has index zero!

Example:

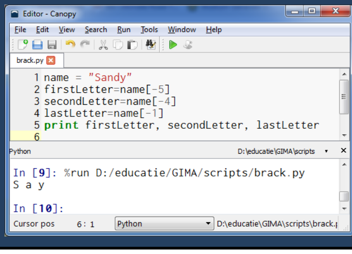


Bracket operator, negative index

```
LETTER = STRING[J]
```

If J < 0:
J = -1 yields the last letter of STRING
J = -2 the letter before, etc.

Example:



Compound data type, syntax of bracket operator (2)

String slice: a segment of a string

Syntax:

SLICE = STRING[I:J]

with:

- STRING, a variable of data type string
- I, index for start of segment, a variable of data type integer
- J, index for end of segment, a variable of data type integer
- SLICE, a segment of STRING (note: SLICE is also of type string)

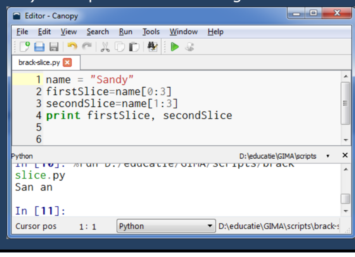
Bracket operator, slices (1)

SLICE = STRING[I:J]

I and J non-negative, J should be greater than I:

SLICE consists of the (I+1)-eth up to and including the J-eth character

Example:



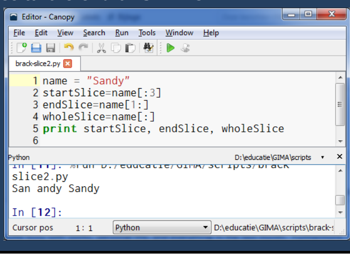
Bracket operator, slices (1)

SLICE = STRING[I:J]

Omitting I: the slice starts at the beginning of STRING

Omitting J: the slice goes to the end of STRING

Example:



Bracket operator, example (1)

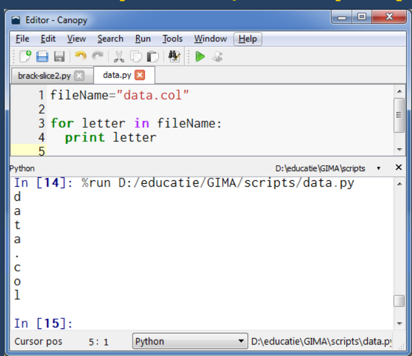
Given: a variable that contains the name of file (e.g. from keyboard input) :

`fileName="data.col"`

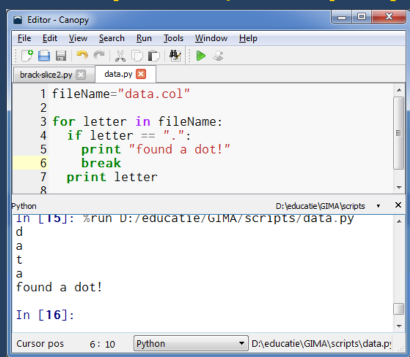
Aim: a program that prints just the basename of the filename

`data`

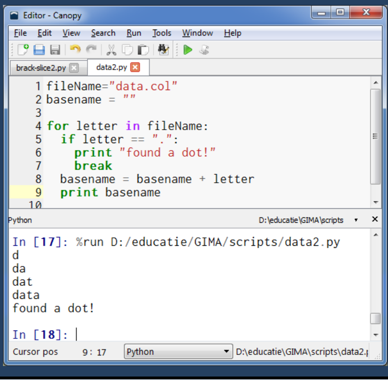
Bracket operator, example (2)



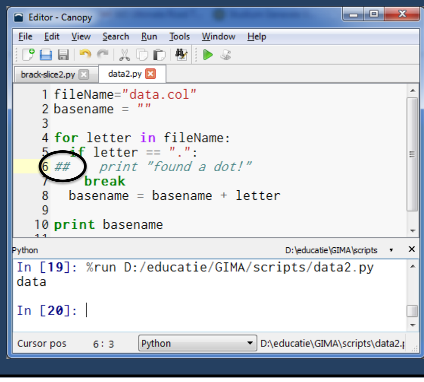
Bracket operator, example (3)



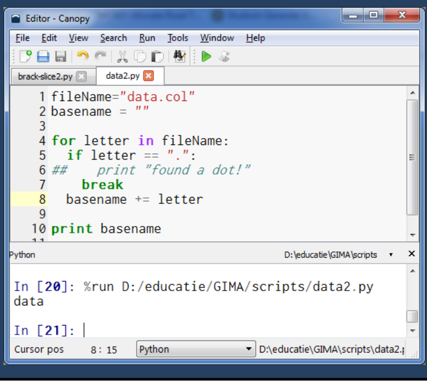
Bracket operator, example (4)



Bracket operator, example (5)



Bracket operator, example (6)



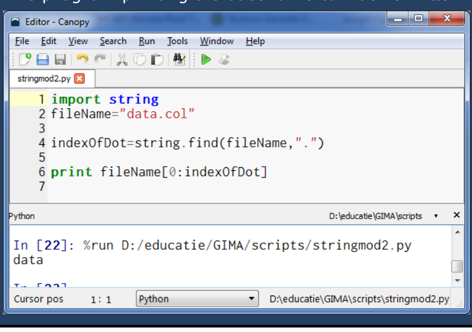
The string module (library)

Contains ('preprogrammed') functions on strings, e.g.:

```
import string

aString="sandY"

capitalize=string.capitalize(aString)       # returns Sandy (a string)
lower=string.lower(aString)                 # returns sandy (a string)
replace=string.replace(aString,"sa","ci")  # returns cindY (a string)
find=string.find(aString,"n")               # returns 2 (an integer),
                                            # index of the letter n
```

Using the string module

The program printing the basename can be rewritten!



Lists

What is a list?

Ordered set of values (compound data type), values are the so-called elements of a list

An element can be 'anything', e.g.

- a string
- a floating-point
- another list
- etc.

Each element is identified by an index

Comparison between strings and lists

Resemblances:

- both consist of elements
- both refer to an element using an index
- both use bracket operator ([]) for referring to elements

Difference:

- string elements are single letters; list elements can be anything

Creating lists

Most often used are:

```
firstList = [0.12, 23.4, 12.5]    # three elements
                                  # of type floating-point

secondList = ["New York", "Amsterdam"] # two elements
                                       # of type string

thirdList = [3, 5, 7, 9]       # four elements
                              # of type integer
```

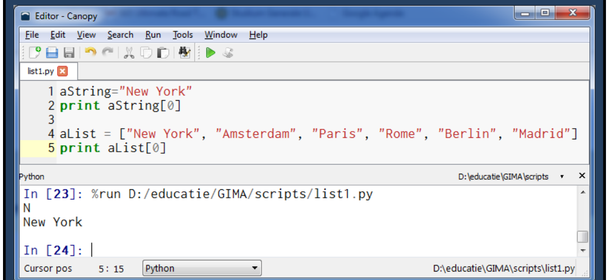
The thirdList can also be created with the range function:

```
thirdList=range(3,10,2)       # the list [3, 5, 7, 9]
```

Accessing single elements

Use bracket operator

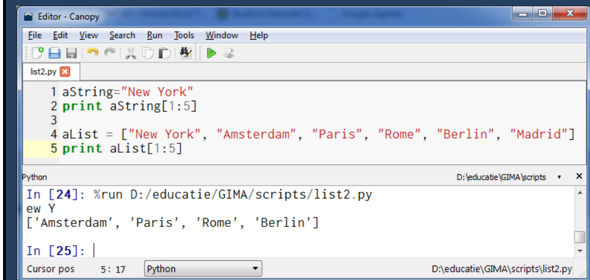
Very similar to accessing elements of a string



Accessing slices

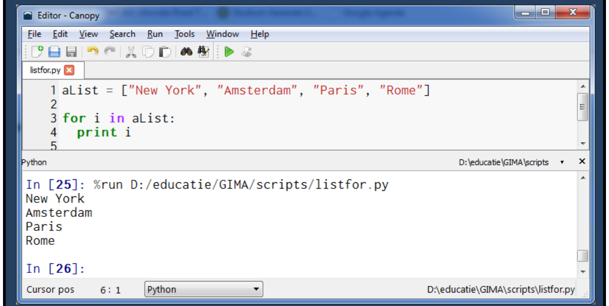
Use bracket operator

Very similar to accessing slices of a string



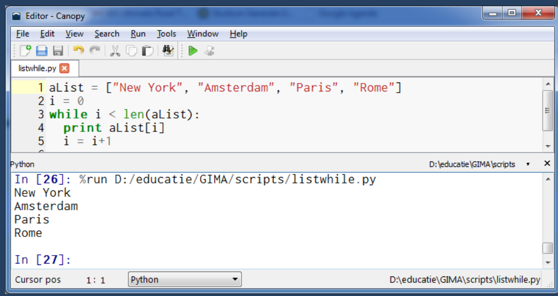
Accessing elements in a loop (1)

With a for loop (shortest):



Accessing elements in a loop (2)

With a while loop:



Strings are unmutable, lists are mutable (1)

Strings are unmutable, i.e. you cannot directly change an element:

```
aString = "Back"
# try to change the "B" to a "J"
aString[0]="J"
```

prints:

```
Traceback (most recent call last):
  File "stringmutable.py", line 3, in ?
    aString[0]="J"
TypeError: object doesn't support item
assignment
```

Strings are unmutable, lists are mutable (2)

Lists are mutable, i.e. you can directly change an element:

```
aList = [0.12, 23.4, 12.5]
# change the first element (0.12) to 2.34
aList[0]=2.34
print aList
```

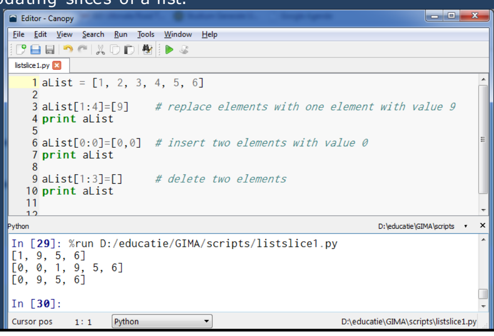
prints:

```
[2.3399999999999999, 23.399999999999999, 12.5]
```

Question: Why is there a rounding error?

Strings are unmutable, lists are mutable (3)

Updating slices of a list:



Nested lists

A list that is an element in another list, e.g,:

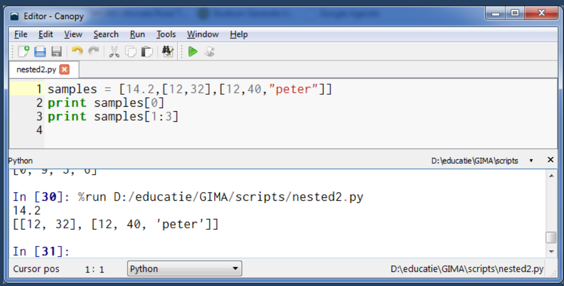
```
samples = [["x","y","z"],[12,32,7],[12,40,7]]
```

All combinations of length of lists and types are possible, e.g.:

```
aList= [14.2,[12,32],[12,40,"peter"]]
```

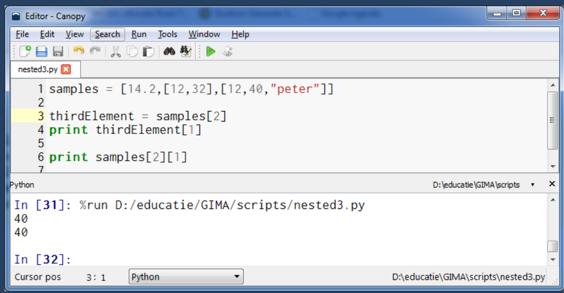
Accessing nested lists

Syntax corresponds to 'normal' lists, e.g.:



Accessing an element in a nested list

Syntax corresponds to 'normal' lists, e.g.:



Accessing all elements in a nested list (1)

We have nested lists:

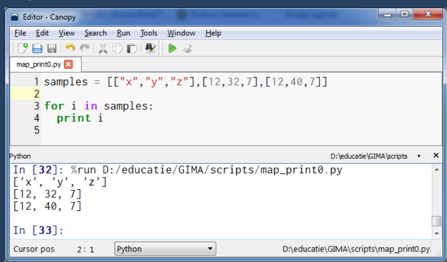
```
samples = [["x","y","z"],[12,32,7],[12,40,7]]
```

Let's make a program that prints each individual value, formatted as a table:

```
x     y     z
12    32    7
12    40    7
```

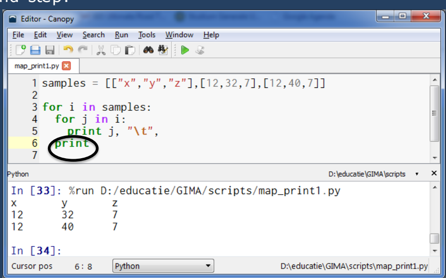
Accessing all elements in a nested list (2)

First step:



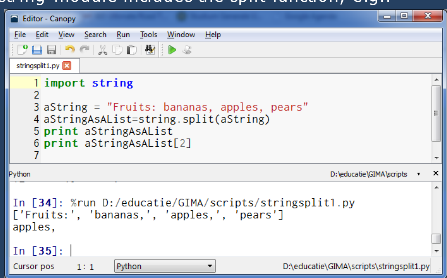
Accessing all elements in a nested list (2)

Second step:



String to list conversion (1)

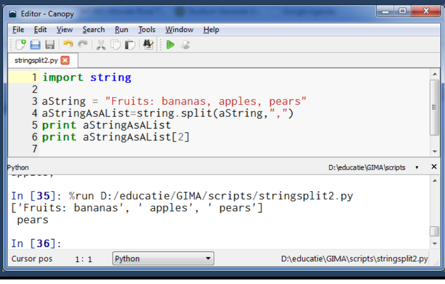
The string module includes the split function, e.g.:



String to list conversion (2)

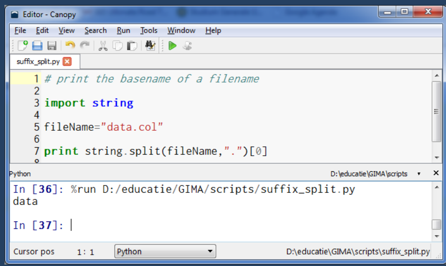
By default split splits at a whitespace character

With an additional argument, other characters can be used for splitting:



String to list conversion (3)

Now, we have another approach to print the basename of a filename:



Files

Computer memory and files

Computer memory

- is used by the program to store data (e.g. variables) while running the program
- disappears when the program ends or the computer shuts down
- is mainly managed by Python (you don't need to do that)

Files

- can be used in a program to open or store specified data
- data are stored permanently
- storage and manipulation needs to be defined in the program (explicitly)

Files: opening and closing

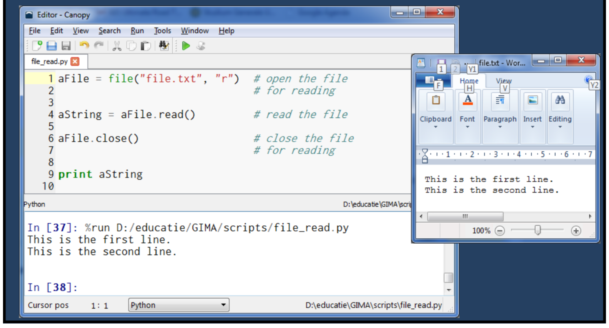
Like with a book, you need to do the following steps to read/write from/to a file:

- open the file
- read from the file
- or write to the file
- close the file

```
f = file("file.txt", "r")  # open an existing file,
                           # "r" indicates opening
                           # for reading

  # read here from the file ....

f.close()                  # close the file
                           # for reading

  # do something without the file or
  # do something else with the file
  # e.g. writing to the file
```

Files: example reading

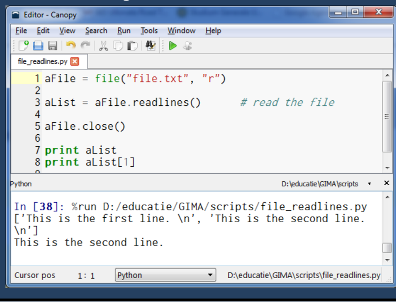
read() returns a string with all contents of the file



Files: example reading

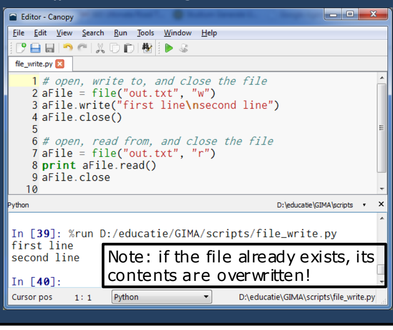
readlines()

- returns a list
- each element is a string with the content of one line from the file



Files: example writing

write() writes a string to a file



Note: if the file already exists, its contents are overwritten!