Compact Course Python

Michaela Regneri & Andreas Eisele

Lecture I

Overview

- What is programming?
- variables
- data types
- values
- operators and expressions
- control structures: `if`, `while`

Programming

- a programmer wants to solve a problem in a systematic way
- an algorithm is an abstract, detailled computing instruction that solves the problem
- a program is is a realization of the algorithm in a specific programming language
- a program can be executed with different inputs

An algorithm for the *maximum number*

- given a list `list` of n integers - we look for the maximum number in `list`
- possible algorithm:
    - store the first number in `list` as current maximum
    - look at every following number one after another
        - compare the currently considered number with the current maximum
        - if the number is greater, change the maximum to the number‘s value
        - after looking at all numbers in `list`, the stored maximum is the maximum number in `list`

Programs ...

- are concrete implementations of an algorithm in a programming language
- use constructs of the programming language to make intuitive concepts of an algorithm precise
  - Loops, conditions, variables, ...
- the exact steps depend on the programming language and its available functions.

Simplest Python Program

the program with this code:

```
print(”Hello, Duckling!“)
```

...outputs:

```
Hello, Duckling!
```

Some technical notes

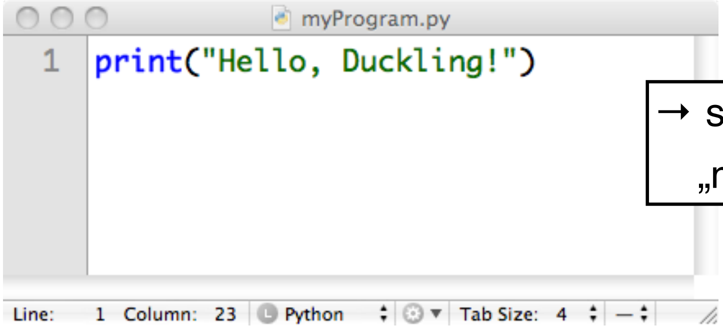
- you can use python *interpreted* or *compiled*
- without going into the technical details, the two possibilities in practice look like this:

interpreted

```
dhcp104-212:~ Michaela$ python

Python 3.1.1 (r311:74543, Aug 24 2009, 18:44:04)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license"
for more information.
>>> print(”Hello, Duckling!“)
Hello, Duckling!
```

Some technical notes



```
print("Hello, Duckling!")
```

→ save as „myProgram.py“

compiled

```
dhcp104-212:~ Michaela$ python myProgram.py
Hello, Duckling!
dhcp104-212:~ Michaela$
```

"maximum number" in Python

- store the first number in `list` as the current maximum number
- check the second to last number in `list`
- if the current number is greater than the previous maximum (`max`), store it as the current maximum
- at the end, the stored value of `max` is the maximum number in `list`

```
max = list [0]
i = 1
while i < len(list):
    if list[i] > max:
        max = list[i]
    i = i + 1
```

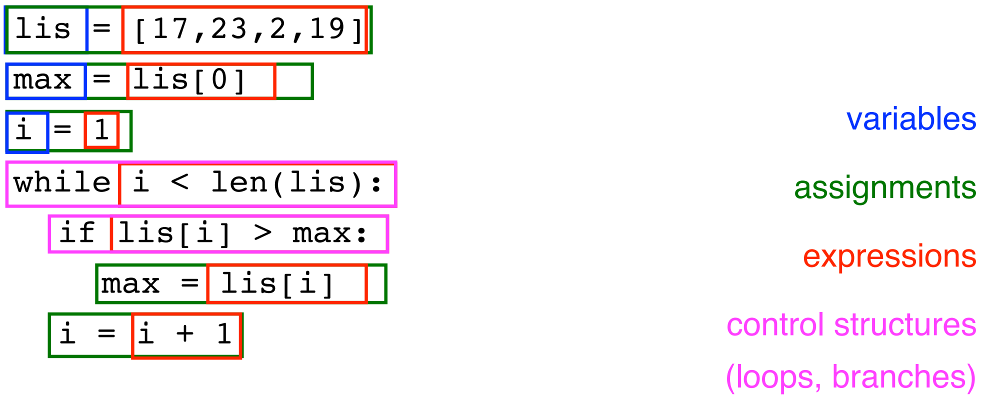
caution: we have ignored the special case of an empty list

Imperative programming

- Python is (basically) an imperative programming language
- programs are sequential lists of instructions
- expressions have values
- values can be assigned to variables
- the main tool to organize the program flow are so-called *control structures*

Elements of imperative programs

- Given a list `lis` of integers, find the greatest number in `lis`.



```
lis = [17,23,2,19]
max = lis[0]
i = 1
while i < len(lis):
   if lis[i] > max:
      max = lis[i]
   i = i + 1
```

Variables, values, data types

- Values in Python may have different data types: numbers, lists, strings, ...
- Variables point to positions of the memory where values are stored
- Dynamic typing: variables don‘t have *fixed* data types
  - The type of a variable is the assigned value‘s type
  - During the program‘s runtime, a variable can take values of different types

Some data types

- Truth values: bool
  (Type of the constants `True` and `False`)
- Numbers: `int`, `long`, `float`, `complex`
- Strings: `str`, `Unicode`
- Collections: `tuple`, `list`, `set`, `dict`
- [...]

Expressions

- Expressions are constructs describing a value
- We distinguish:
  - Literals: Expressions from / in which the value can be directly read / written
  - Variables
  - complex expressions with operators
  - calls of functions or methods

Integers

- `int` (plain integers)
  - Value range: -2^b, ..., $+2^{b-1}$, $B \geq 31$ (system dependent)
- `long` (long integers)
  - (in Python:) arbitrarily large integers
- Integer literals ($i = 3$)
  - denote values of type `int`
  - Exceptions: The number exceeds the range of acceptable values, or the literal ends with "L"

Integer literals

- "standard" numbers written as `17`, `0`, `-23` in the source code are interpreted as decimal (base 10)
- Literals starting with `0o` (or `0O`) are interpreted as octal (base 8) integers
  (Example: `0o13` represents value `11`)
- Literals that start with `0x` are interpreted as a hexadecimal (base 16) integers
  (Example: `0x1ca` represents `458`)

Floating point numbers

- Floats are represented as decimal numbers (1.1, 47.11)
- Range depends on the system
- Often, the internal representation of decimal numbers is imprecise

```
>>> 0.1
0.1
>>> 0.1 * 10000000000000000000000000000000
1.0000000000000001e +31
```

Operators

- Elementary (arithmetic) operations are represented by operators.
- Arithmetic operators (selection):

```
a + b
a - b
a * b
a / b
a % b
```

(Modulo / remainder with integer division)

- If a, b do not have the same type, the operations result in a value of the more general type

Precedence

- an expression may contain more than one operator: `2 * 3 + 4`
- The order in which the operators are evaluated is called *Precedence*
- With parentheses, precedence can be indicated directly:

```
>>> (2 * 3) + 4
10
>>> 2 * (3 + 4)
14
```

Precedence

- Without parentheses, standard precedence rules are applied (multiplication/division before addition/subtraction): `2 * 3 + 4 = (2 * 3) + 4`
- a question of style: somtimes it is recommendable to use parentheses even if they are redundant (legibility)
- Don‘t use parentheses when precedence is irrelevant: `2 + 3 + 4` is better than `2 + (3 + 4)`

Relational operators

- relational operators:

```
a < b    a > b
a <= b   a >= b
a == b   a != b
```

(less / greater)

(greater than or equal to)

(equal or not equal)

- The result of such a comparison is a boolean (bool)

```
>>> 3 > 2
True
>>> (2 * 3) + 4 !=  2 * 3 + 4
False
```

Truth values

- The type `bool` represents the two truth values `True` and `False`
- Operations:
  - `not a` (Negation)
  - `a and b` (Conjunction)
  - `a or b` (Disjunction)
- Precedence: **not** ⋙ **and** ⋙ **or**

  `a and not b or c = (a and (not b)) or c`
- *Short-circuit evaluation*: the evaluation stops as soon as the result is evident (.: True or something).

String literals

```
'This is a String.'

"That, too."

"He said \" Hello \"."

'He said "hello". '
```

- Note: strings may not contain any special characters (umlauts etc. etc.) if no encoding is specified.
- encoding is specified in the first code line:

```
# -*- Coding: utf-8 -*-
# -*- Coding: latin-1 -*-
```

String operators (selection)

- Concatenation:

```
>>> 'Hello' + 'World'
'HelloWorld'
```

- Access to individual characters with list indices:

```
>>> 'Hello'[0]
'H'
```

```
>>> 'Hello'[1]
'e'
```

- Test whether a substring occurs:

```
>>> 'He' in 'Hello'
True
```

```
>>> 'Ha' in 'Hello'
False
```

String operators (selection)

- Length:

```
>>> len('Hello')
5
```

- Convert to a different data type (number):

```
>>> int('123')
123
```

```
>>> float('123')
123.0
```

Variables

- one can assign the value of an expression to variables
- variables can be evaluated in order to use their value in an expression

```
>>> number = 123
>>> number = number + 2
>>> print(number)
125
```

- print is a function that prints the value of an expression to the screen (actually: the standard output)

Variables

- variables (more generally, all identifiers) must start with a letter or "_". The remainder may include digits.
- umlauts etc. are not allowed (ASCII encoding)
- the name must not be a keyword (`if`, `while`, etc.)
- the names are case-sensitive
- Examples:
  √ OK: `Foo, foo12, _foo`
  X wrong: `2foo, if, überzwerg`

Assignments

- `var = expr` the expression `expr` is evaluated, then its value is stored in `var`.

- $\texttt{var}_1 = \texttt{Var}_2 = \ldots = \texttt{expr}$
  the value of `expr` is assigned to all $\texttt{var}_i$

- $\texttt{var}_1, \ldots, \texttt{var}_n = \texttt{expr}_1, \ldots, \texttt{expr}_n$
  - all $\texttt{expr}_i$ are evaluated, then the corresponding values are assigned to $\texttt{var}_i$
  - Example: `a, b = 'a', 'b'`

Assignments

- Assignments of the form `x = x + y` are very common: the value of a variable `x` is combined with another value and then immediately re-assigned to `x`
- Shorthand syntax:

```
x += expr
x -= expr
x *= expr
x /= expr
x %= expr
```

Statements

- a Python program is a sequence of *statements*
- Seen so far: assignments, **`print`**
- a statement roughly corresponds to a step in the underlying algorithm
- statements are separated by line breaks: each line is (usually) exactly one statement
- it is possible to separate (short) statements with semicolons (and write them in the same line)

Control Structures

- Sometimes one wants to execute statements repeatedly, or only under certain conditions
- This is the purpose of control structures
  - conditions: **`if`**
  - loops: **`while`**, **`for`**

if - else

- if $\texttt{expr}_1$ evaluates to **`True`**, $\texttt{block}_1$ will be executed.
- otherwise $\texttt{block}_2$ will be executed.
- Values evaluating to **`False`**: **`False`**, 0, empty string, empty list, empty sets, ...
- All other values evaluate to **`True`**

`if` $\texttt{expr}_1$**:**
  $\texttt{block}_1$
[**`else`**:
  $\texttt{block}_2$]

a „block“ consists of one or more statements (~lines)

if - elif - else

- expressions are evaluated in the given order, until one is found to be **True**
- then the corresponding block is executed.
- If none of the expressions is true, the **else** block is executed (in case there is one)

```
if expr1:
  block1
[elif expr2:
  block2]
...
[else:
  blockk]
```

Indentation

- Spaces are important: blocks of a if-statement must be indented!

```
if a < b:
  if a < c:
    print("foo")
  else:
    print("bar")
```

```
if a < b:
  if a < c:
    print("foo")
else:
  print("bar")
```

Blocks

- several statements can be grouped into a *block* by indenting the respective statements equally
- Instructions in the same block have the same number of the same type of whitespace character
- best practice: always stick to one type of whitespace character (either tab or space)

```
if a < 10:
  print("foo")
  a = a + 1
```

while

1. The expression `expr` is evaluated.
2. If it evaluates to `True`, `block` is executed. After that, go to 1.
3. Otherwise, the program flow resumes after the loop (next statement with same indent as „`while`“)

```
while expr:
  block
```

Greatest common divisor

- The greatest common divisor of two integers m and n is the largest integer by which both m and n are divisible without remainder
- Euclidean algorithm: in each step, a division with remainder is done. In the next step, the remainder is the new divisor.
- The first divisor giving a remainder of 0 is the greatest common divisor of the two input numbers

Greatest common divisor

- Example: Calculate the greatest common divisor of 1071 and 1029

```
1071 / 1029 = 1, remainder: 42
1029 / 42 = 24,  remainder: 21
  42 / 21 = 2,   remainder: 0
```

- Thus, 21 is the greatest common divisor of 1071 and 1029

Greatest common divisor in Python

- the variables `x` and `y` contain the input numbers
- when the coputation finishes, the variable `g` stores the greatest common divisor of `x` and `y`.

```
g = y
while x > 0:
   g = x
   x = y%x
   y = g
```

break & **continue**

- The **break** statement exits the current loop without evaluating the condition
- The **continue** statement skips the remainder of the current iteration, evaluates the condition again and continues the loop (if the condition is True)

while - else

- loops may have **else**-statements
- the **else**-statements is executed as soon as the loop‘s condition evaluates to false...
- ...but *not* if the loop was aborted by a break statement

Example: prime numbers from 2 ... 100

```
n = 2
while n < 100:
   m = 2
   while m < n:
      if n%m == 0:
         break
      m += 1
   else:
      print(n, 'is a prime number')
   n += 1
```

Summary

- expressions are constructs that have a value
- values have types.
- variables are expressions to which values can be assigned
- with `if`-statements, you can decide at runtime which parts of a program shall be executed.
- with `while` loops, the same statement can be executed repeatedly (under certain conditions)

Command line arguments

- you can pass *command line arguments* to Python programs
- the i-th argument is accessed with `sys.argv[i]` (count starts at 1); the value is a string

file: echo.py

```
import sys
print(sys.argv[1])
```

- `python echo.py bla blub` ⇒ output: `bla`