Introduction to programming
Lecture 2: Basic programming

UNIVERSITY OF
GOTHENBURG

Richard Johansson

September 8, 2015

overview

Overview of this lecture

- recap of previous lecture: basic programming
- lists
- repetition
- conditions
- functions

overview

introduction

values, expressions, variables, statements

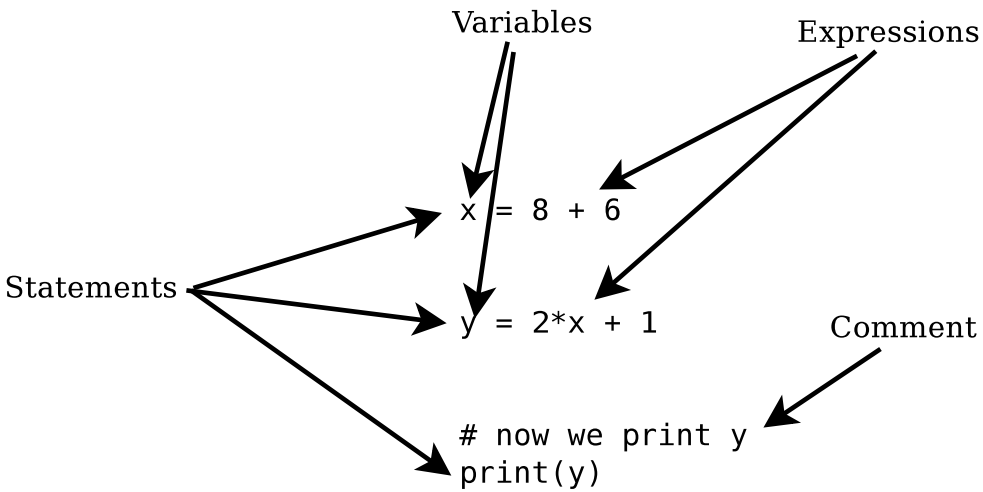
repetition

conditionals

functions and methods

basic programs

- **expressions** compute **values**
- values have **types**
- **variables** remember values
- a program is a sequence of **statements** making use of expressions
- the Python interpreter **executes** the statements sequentially



example with Pythontutor

```
x = 8 + 6

y = 2*x + 1

print(y)
```

numbers

- `int`: integers (whole numbers) such as 5, 1, -7
- `float`: floating-point numbers such as 3.14, 5.0, -10.7
- we can use basic arithmetic: `+ - * / **`

- int + int → int, int * int → int, etc
  - BUT int / int → float (in Python 3)
- float + float → float etc
- float + int → float etc

strings

- a **string** is a piece of text
- strings are written with single or double quotes
  - but the quotes are not actually included in the string: they are there to show where the string begins and ends
- multiline strings can be written with three double quotes (""")
- examples:

```
s1 = 'a string'
s2 = "another string"
s3 = """a long string spanning
more than one line"""

print(s1)
```

check

```
s1 = "abc"

s1 = abc

s1 = 'abc'

s1 = 'abc"

s1 = '  abc '
```

string "arithmetic"

- the + sign has a special meaning for strings: concatenation of two strings ("gluing")
- the * sign is used to copy a string multiple times
- examples:

```
s1 = 'abc'
s2 = "def"
s3 = s1 + s2
print(s3)

s4 = s1 * 5
print(s4)
```

substrings

- we can access a part of the string by using index notation [ ]
- `s[k]` gives us the letter at position `k` **starting at 0**
- example:

```
s = 'this is a string'
print(s[2])
```

- `s[j:k]` gives us the part of the string starting at position `j` up to the position `k` **but not including `k`**
  - in Python terminology, this is called **slicing**

```
print(s[5:9])
```

- similarly:

```
print(s[5:])
print(s[:9])
```

lists

- lists are used to represent sequences of data, e.g., the words occurring in a document
- in Python, they are written using square brackets `[ ]`
- example: `['Python', 'programming']`
- parts of a list can be accessed just as for strings – indexing and slicing

```
l = [12,43,564,1,23]
print(l[4])
print(l[1:3])
```

- unlike strings, lists can be modified:

```
l[3] = 88
print(l)
```

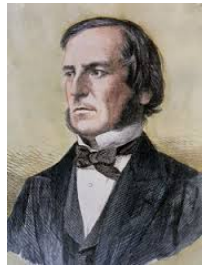
a slightly more complicated example

```
mylist = [7, 8, 4, 3]

mylist[0] = 650

mylist[2] = [86, 45]

mylist[1] = [120]

mylist[4] = 1000

print(mylist)
```

truth values: Booleans

- the Boolean (`bool`) type is used to represent logical truth values
- there are two possible values: `True` and `False`
- Boolean values often come from e.g. comparisons, and they are used in conditional statements (`if`)

```
x = 100
y = 150
z = 100
truthvalue1 = x < y
truthvalue2 = x < z
print(truthvalue1)
print(truthvalue2)
```



a special value: `None`

- the special value `None` is used to represent “empty” results
- its type is called `NoneType`
- more about this later!

types: summary

int	5 -7 0 48
float	5.0 3.2 0.0 -6.7
str	"a string" "abc" " " ""
bool	True False
NoneType	None
list	[12, 41, 8] ["a", "list"] ["s", 5] []

overview

things to do with a list of numbers

- what is the sum of the numbers in a list?

Example: sum the numbers in a list

```
mylist = [7, 4, 8, 12]

listsum = 0
listsum = listsum + mylist[0]
listsum = listsum + mylist[1]
listsum = listsum + mylist[2]
listsum = listsum + mylist[3]

print(listsum)
```

- how to do this for a long list?
- how to do this for a list where we don't know the length in advance?

Example: sum the numbers in a list (cont)

- set initial value of sum to 0
- for every item x in the list:
  - add x to the current value of the sum
- print the sum

The **for** statement: repetition

- to convert the idea expressed on the previous slide into Python code, we use the **for** statement
- do something for each member of a collection (list, string, . . .)
- in programming jargon, doing something repeatedly is called a **loop**

sum the numbers in a list (properly)

```
mylist = [7, 4, 8, 12]

listsum = 0
for x in mylist:
    listsum = listsum + x

print(listsum)
```

indentation of blocks

- to show which statements are to be executed for each step in a `for` loop, we **indent** those statements: put them a bit to the right of the start of the `for`
- we say that they are in a separate **block**

```
listsum = 0
for x in mylist:
    listsum = listsum + x
print(listsum)
```

proper indentation is important!

```
mylist = [7, 4, 8, 12]
listsum = 0
for x in mylist:
    listsum = listsum + x
print(listsum)
```

```
mylist = [7, 4, 8, 12]
listsum = 0
for x in mylist:
    listsum = listsum + x
    print(listsum)
```

doing something 10 times

```
steps = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for step in steps:
    print(step)
```

```
for step in range(10):
    print(step)
```

summing a list of lists of numbers

```
mylist = [[7, 8], [9, 6, 2], [1, 5, 2, 9], [3]]
listsum = 0
for sublist in mylist:
    for x in sublist:
        listsum = listsum + x
print(listsum)
```

things to do with a list of numbers (again)

- how many numbers are there in the list?

overview

printing the largest of two numbers?

- user gives us two numbers x and y
- how can we print a message saying which of them is the largest (or whether they are equal), e.g.

```
x is the largest
```

or

```
y is the largest
```

or

```
x and y are equal
```

doing different things depending on a condition

- The if statement will execute a block depending on a condition
- Simplest case:

```
if x > 1000:
    print("x is greater than 1000")
```

- Selecting one of two alternatives:

```
if x > 1000:
    print("x is greater than 1000")
else:
    print("x is not greater than 1000")
```

- Even more alternatives:

```
if x > 1000:
    print("x is greater than 1000")
elif x < 0:
    print("x is less than 0")
else:
    print("x is between 0 and 1000")
```

printing the largest of two numbers

- assume we are given x and y . . .

```
if x > y:
     print("x is the largest")
elif x < y:
     print("y is the largest")
else:
     print("x and y are equal")
```

conditions involving numbers

`<`	less than
`<=`	less than or equal to
`==`	equal to (note: two "=" signs, not one)
`!=`	not equal to
`>`	greater than
`>=`	greater than or equal to

The result of each of these tests is a `bool`: `True` or `False`.

conditions involving strings

==	equal to
<	alphabetically before
t in s	test if t is contained inside s

combining conditions

- **not** *CONDITION*
- *CONDITION1* **and** *CONDITION2*
- *CONDITION1* **or** *CONDITION2*

things to do with a list of numbers (again)

- which is the largest number in the list?

one possible solution

```
mylist = [7, 4, 8, 12]
maximum = mylist[0]
for x in mylist:
    if x > maximum:
        maximum = x
print(maximum)
```

overview

summing three lists

- we have three lists and want to print the sum of each of them
- how to avoid this repetition?

```
mylist1 = [7, 4, 8, 12]
listsum1 = 0
for x in mylist1:
    listsum1 = listsum1 + x
print(listsum1)

mylist2 = [3, 9, 11, 17, 6]
listsum2 = 0
for x in mylist2:
    listsum2 = listsum2 + x
print(listsum2)

mylist3 = [21, 16]
listsum3 = 0
for x in mylist3:
    listsum3 = listsum3 + x
print(listsum3)
```

functions

- a **function** is a part of the program put separately
- we **call** the function and supply **inputs** to it
- it will carry out its computations and possibly **return** an **output**
- examples:

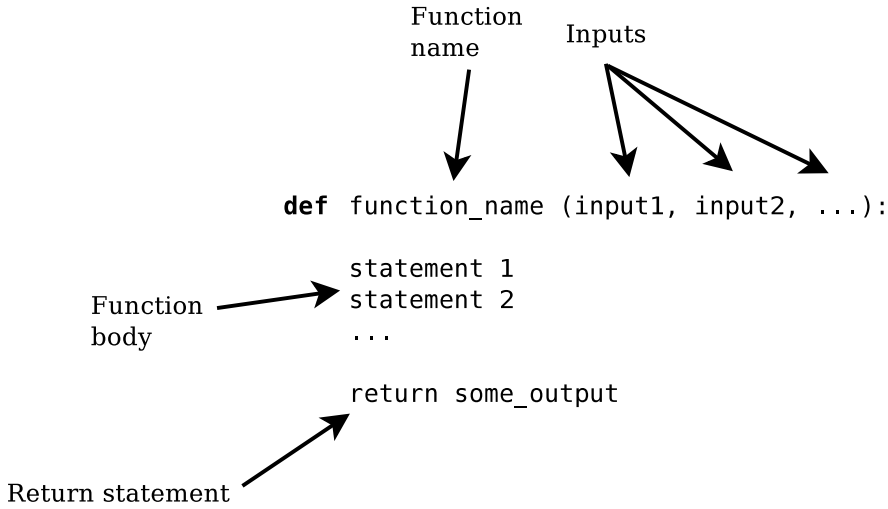
```
l = len('this is a short text')

print('the length of the string is', l)
```

built-in and user-defined functions

- the functions `len` and `print` are examples of **built-in** functions: they are part of the Python language
- we can also make our own **user-defined** functions
- benefits of declaring functions:
  - avoiding repetition
  - improving readability of your program by splitting it into logically separated parts

declaring functions

- the keyword **def** is used to declare a function



- examples:

```
def compute_house_area(length, width):
    area = length*width
    return area

def print_help_message():
    print("Please consult the manual!")
```

a special case: functions returning no value

```
def print_help_message():
    print("Please consult the manual!")
```

- if there is no `return` statement, then the special value `None` is implicitly returned

summing three lists: better

```
def sum_list(mylist):
    mysum = 0
    for x in mylist:
        mysum = mysum + x
    return mysum

mylist1 = [7, 4, 8, 12]
print(sum_list(mylist1))

mylist2 = [3, 9, 11, 17, 6]
print(sum_list(mylist2))

mylist3 = [21, 16]
print(sum_list(mylist3))
```

- have a look in Pythontutor...

actually, we are reinventing the wheel...

- Python has a number of **built-in** functions:

```
mylist = [7, 4, 8, 12]

print(len(mylist))

print(sum(mylist))

print(max(mylist))

print(min(mylist))
```

methods

- values in Python can have their own functions
- these functions are called **methods**
- we call the method `m` on the value `x` like this: `x.m(inputs)`

```
s = "here is a string"

modified = s.replace("i", "I")

print(modified)
```

some methods on strings

s.lower()	gives a lowercased copy of s
s.startswith(t)	test whether s starts with t
s.endswith(t)	test whether s ends with t
s.islower()	test if all cased characters in s are lowercase
s.count(t)	counts the number of occurrences of t in s
s.split(t)	splits s into a list of substrings
s.replace(f, t)	gives a copy of s where f is replaced by t
	...

See `http://docs.python.org/3/library/stdtypes.html`

assignment 1

- Using WordNet in NLTK
- `https://svn.spraakdata.gu.se/repos/richard/pub/itp2015_web/assign1.html`
- deadline: **September 18**

a quick note about modules (for assignment)

```
import random

random_number = random.randint(0, 10)
print(random_number)
random_number = random.randint(0, 10)
print(random_number)
```