Introduction to programming
Lecture 3

UNIVERSITY OF GOTHENBURG

Richard Johansson

September 15, 2015

today's lecture

- last time we saw many new concepts, so today we repeat a bit
  - lists
  - repetition (iteration) through lists with `for`
  - conditions with `if`
  - functions
- also a bit of new material: mostly to fill in the missing pieces for the assignment

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']`
- indexing and slicing to give a part of the list:

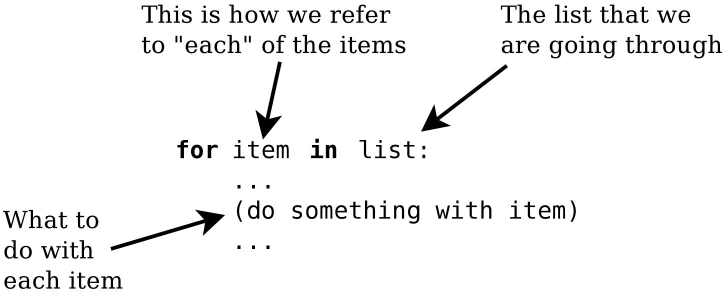
```
l = [12,43,564,1,23]
print(l[4])
print(l[1:3])
```

- unlike strings, lists can be modified:

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

going through a list: iterating using **for**

- do something for each member of a collection (list, string, . . .)
- in programming jargon, doing something repeatedly is called a **loop**



This is how we refer to "each" of the items

The list that we are going through

```
for item in list:
    ...
    (do something with item)
    ...
```

What to do with each item

example: sum the numbers in a list

```
numbers = [18, 7, 4, 8, 12, 5]

listsum = 0
for number in numbers:
    listsum += number

print(listsum)
```

note: we might as well have written `sum(numbers)`

a list trick: list comprehension

- if we have a list `l`, we might want to create a new list where we have applied some operation to all members of `l`
- examples
  - `["a", "sentence"]` → `["A", "SENTENCE"]`
  - `[1.2, 7.5, 3.15]` → `[1, 7, 3]`
- list comprehension does the trick:

```
words = ["a", "short", "sentence"]
capitalized_words = [ word.upper() for word in words ]

floats = [1.2, 7.5, 3.15]
rounded = [ int(number) for number in floats ]
```

strings

- a **string** is a piece of text
- in the code, we write them with quotes (single, double, """ for multiline)
- we can use the + and * signs to concatenate or repeat:

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

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

string tricks: splitting and joining

- the string method `split` splits a string into a list of strings
  - `s.split()` splits at spaces
  - `s.split(separator)` splits at `separator`
- conversely, `join` takes a list of strings and puts them together with a `separator` in between
  - `separator.join(strings)`
- example:

```
sentence_string = "this is a sentence"
words = sentence_string.split()
for word in words:
    print(word)
new_sentence_string = "_".join(words)
print(new_sentence_string)
```

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`

UNIVERSITY OF
GOTHENBURG

string tricks: string formatting

- in some cases we want to format several outputs in a nice, structured way
- with **string formatting**, we create a string template and insert variables into it
    - easier than concatenating strings with +
- also for printing fixed-width columns
- Python has two different styles of string formatting:
    - using %
    - using `format`

```
gross_salary = 25000
tax_rate = 0.3125
tax = gross_salary * tax_rate
net_salary = gross_salary - tax

print("gross: %s, tax: %s, net: %s" % (gross_salary, tax, net_salary))
```

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])
```

converting between types

str(x) makes a string

- `str(5)` gives `"5"`
- `str(5.4)` gives `"5.4"`

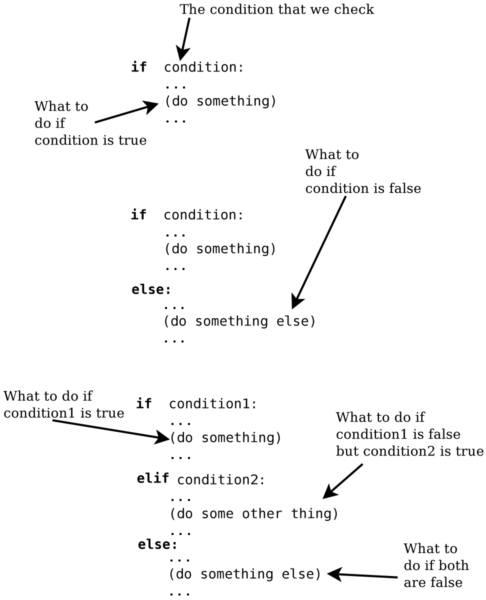
int(x) makes an integer number

- `int("5")` gives 5
- `int(5.4)` gives 5

float(x) makes a floating-point number

- `float("5.4")` gives 5.4
- `float(5)` gives 5.0

the if statement



The condition that we check

What to do if condition is true

```
if  condition:
    ...
    (do something)
    ...
```

What to do if condition is false

```
if  condition:
    ...
    (do something)
    ...
else:
    ...
    (do something else)
    ...
```

What to do if condition1 is true

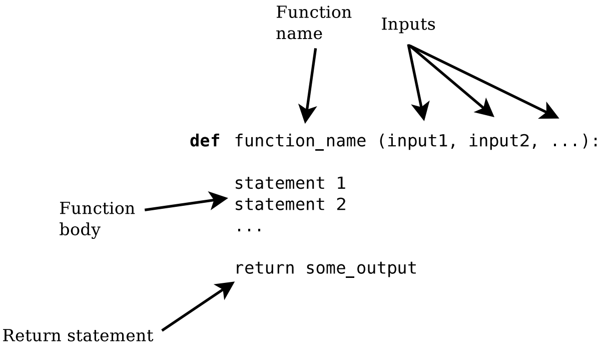
What to do if condition1 is false but condition2 is true

What to do if both are false

```
if  condition1:
    ...
    (do something)
    ...
elif condition2:
    ...
    (do some other thing)
    ...
else:
    ...
    (do something else)
    ...
```

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 **return** an **output**
- benefits of declaring functions:
  - avoiding repetition
  - reusing later
  - improving readability



Function name

Inputs

```
def  function_name (input1, input2, ...):

     statement 1
     statement 2
     ...

     return some_output
```

Function body

Return statement

example functions

```
euro_rate = 9.33156
yen_rate = 0.0689876

def kr_to_euros(kr_amount):
    euro_amount = kr_amount / euro_rate
    return euro_amount

def euros_to_kr(euro_amount):
    kr_amount = euro_amount * euro_rate
    return kr_amount
```

example: printing the words in a sentence line by line

```
sentence = "this is a sentence"
print_sentence(sentence)
```

example output:

```
0: this
1: is
2: a
3: sentence
```

example: finding the longest word in a sentence

```
sentence = "this is a sentence"
long_word = find_longest_word(sentence)
print(long_word)
```

should print:

```
sentence
```

recall: the built-in function `len` returns the length of a string

functions again: local and global variables

- variables written inside a function are called local variables
    - they are alive only when the program enters a function
- variables written outside all functions are called global variables
    - they are alive during the whole life of the program

```
euro_rate = 9.33156
yen_rate = 0.0689876

def kr_to_yen(kr_amount):
    yen_amount = kr_amount / yen_rate
    return yen_amount

def yen_to_kr(yen_amount):
    kr_amount = yen_amount * yen_rate
    return kr_amount
```

functions calling other functions

```
euro_rate = 9.33156
yen_rate = 0.0689876

def kr_to_yen(kr_amount):
    yen_amount = kr_amount / yen_rate
    return yen_amount

def yen_to_kr(yen_amount):
    kr_amount = yen_amount * yen_rate
    return kr_amount

def euros_to_yen(euro_amount):
    kr_amount = euros_to_kr(euro_amount)
    yen_amount = kr_to_yen(kr_amount)
    return yen_amount

def yen_to_euros(yen_amount):
    kr_amount = yen_to_kr(yen_amount)
    euro_amount = kr_to_euros(kr_amount)
    return euro_amount
```

modules

- we can use **modules** to arrange groups of functions into logically separate parts
  - e.g. a module with our functions to convert currencies
- Python comes with a large number of built-in modules
- you can download modules from the web (e.g. NLTK)
- and you can write new modules yourself
- when telling the program to use a module, we say that we **import** it

importing from a module

- there are different ways to import from a module

```
import math
print(math.sqrt(100))
```

```
from math import sqrt
print(sqrt(100))
```

some useful builtin modules

- re: regular expressions for text processing
- calendar and datetime: handling dates and times
- math: mathematical functions such as cos, exp, sqrt
- pickle: writing Python data to a file
- random: generating pseudorandom numbers
- . . . and many more: see http://docs.python.org/3/library/index.html

random numbers

```
import random

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

importing your own module

- if we have a file called currencies.py, we can import it from another program:

```
import currencies
print(currencies.kr_to_euros(100))
```

importing WordNet from NLTK

```
import nltk.corpus
dog_synsets = nltk.corpus.wordnet.synsets("dog")
```

or

```
from nltk.corpus import wordnet
dog_synsets = wordnet.synsets("dog")
```

or

```
from nltk.corpus import wordnet as wn
dog_synsets = wn.synsets("dog")
```

documenting your programs

- it is important to document the programs you write
- other people may have to read your code
- . . . and you may return after a year!
- in Python, two main ways to document code: **comments** and **docstrings**

comments example

- comments are mainly used for **internal** documentation, where you say **how** your program works

```
euro_rate = 9.33156
yen_rate = 0.0689876

def kr_to_euros(kr_amount):
    # first we compute the amount in kronor by dividing by the
    # euro exchange rate
    euro_amount = kr_amount / euro_rate

    # now we return the amount in euros
    return euro_amount
```

docstrings

- docstrings are strings placed in the beginning of a module or function
- they are used for **external** documentation: saying **what** a program does

```
"""This module contains functions that convert currencies."""

euro_rate = 9.33156
yen_rate = 0.0689876

def kr_to_euros(kr_amount):
    """Convert a given amount in Swedish kronor to euros."""
    return kr_amount / euro_rate

def euros_to_kr(euro_amount):
    """Convert a given amount in euros to Swedish kronor."""
    return euro_amount * euro_rate
```

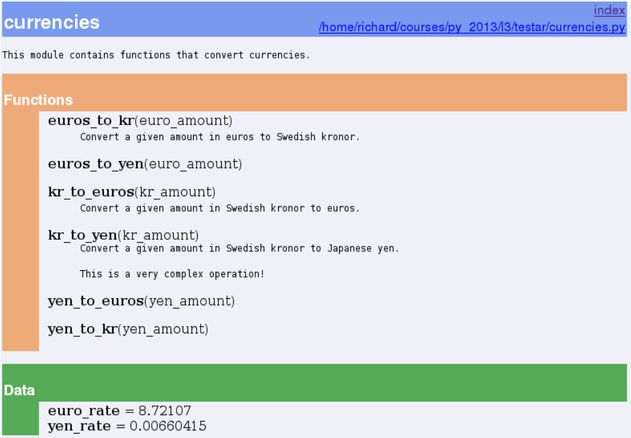
docstrings and the Python interpreter

- if you run the Python interpreter interactively, you can use the `help` command on a module or function name
- it will then print the docstring for that module or function

```
Python 2.7.3 (default, Jan  2 2013, 13:56:14)
>>> import currencies
>>> help(currencies)
>>> help(currencies.kr_to_euros)
```

generating module documentation pages

- we can use the pydoc tool to make documentation web pages
- e.g. `pydoc -w currencies`



currencies

index
/home/richard/courses/py_2013/l3/testar/currencies.py

```
This module contains functions that convert currencies.
```

Functions

euros_to_kr(euro_amount)

```
Convert a given amount in euros to Swedish kronor.
```

euros_to_yen(euro_amount)

kr_to_euros(kr_amount)

```
Convert a given amount in Swedish kronor to euros.
```

kr_to_yen(kr_amount)

```
Convert a given amount in Swedish kronor to Japanese yen.

This is a very complex operation!
```

yen_to_euros(yen_amount)

yen_to_kr(yen_amount)

Data

euro_rate = 8.72107
yen_rate = 0.00660415

user-defined types or **classes**

- programmers can define their own types
  - user-defined types are called **classes**
- for instance, NLTK defines many classes
- you are already using one such class in the exercise: `Synset`
- in later lectures, you will see how to define your own classes

objects, attributes, methods

- values of a class are called **objects**
- an object may contain its own variables: they are called **attributes**
- as we have seen, they may also have their own functions: **methods**
- attributes and methods are accessed with dot notation:
  - `x.attr`
  - `x.method(inputs)`

a look at `Synset`

if `ss` is an object of the class `Synset`:

- `ss.hypernyms()`: a list of more specific kinds
- `ss.hyponyms()`: a list of more specific kinds

- `ss.lemmas`: list of words corresponding to this concept
- `ss.definition`: definition of the concept
- `ss.examples`: list of examples

sets

- **sets** are collections where each item appears only once

```
word_set = set(["this", "is", "a", "set", "of", "words"])

print(word_set)
word_set.add("another")
word_set.add("this")
print(word_set)
```

sets and lists...

```
word_set = set(["this", "is", "a", "set", "of", "words"])
word_list = ["this", "is", "a", "list", "of", "words"]
print(word_set)
print(word_list)
print("words" in word_set)
print("words" in word_list)
```

- differences between sets and lists:
  - lists can have multiple identical items
  - lists remember the order of insertion
  - sets are faster for some operations, e.g. membership testing
  - no indexing or slicing for sets
- similarities between sets and lists:
  - we can use `for` to go through the members
  - `len` gives the size of the set
  - membership test: `x in s`
- converting: `list(s)` and `set(l)`

next lecture: counting words

- recall this example from the first lecture!
- which are the missing pieces?

```
with open("göteborgsposten.txt") as f:
  table = {}

  for line in f:
     for word in line.split():
        if word in table:
           table[word] += 1
        else:
           table[word] = 1

print(max(table, key=table.get))
```