Lecture 2: Classes, Modules, Namespaces

PCL II, CL, UZH

March 02, 2016

Universität Zürich[UZH]

Overview

Topics:

- Functions
- (Types)
- Classes
- Modules
- Namespaces and scope of variables

Aims:

- Understand functions/classes/modules
  - What they are used for
  - How to apply them
  - How to write them

- Understand namespaces/scope of variables:
  - Which region of the code can access which variables
  - Understand what *dotted syntax* is and what is has to do with namespaces

Overview

Universität Zürich[UZH]

Function:

A block of code that can be called as a single instruction from within the program.

Class:

A “plan of construction” for a data type. From this plan, an infinite number of objects of this data type can be created/instantiated (instances).

Module:

A python file whose code can be imported and used by another python file.

Functions: overview

Universität Zürich[UZH]

Functions:

- motivation
- usage
- arguments
- return statements
- (recursion)
- built-in functions

Functions: motivation

```
tokenListA = ["Process", "me", ",", "PLEASE", "!"]
tokenListB = ["ME", "TOO", ",", "please", "do", "me", "next"]

lcaseListA = []

for tokA in tokenListA:
   lcaseListA.append(tokA.lower())



lcaseListB = []

for tokB in tokenListB:
   lcaseListB.append(tokA.lower())


print " ".join(lcaseListA)
print " ".join(lcaseListB)
```

Functions: motivation

```
tokenListA = ["Process", "me", ",", "PLEASE", "!"]
tokenListB = ["ME", "TOO", ",", "please", "do", "me", "next"]

lcaseListA = []

for tokA in tokenListA:
   lcaseListA.append(tokA.lower())


lcaseListB = []

for tokB in tokenListB:
   lcaseListB.append(tokA.lower())

print " ".join(lcaseListA)  # process me , please !
print " ".join(lcaseListB)  # ...
```

Functions: motivation

Universität Zürich[UZH]

```
tokenListA = ["Process", "me", ",", "PLEASE", "!"]
tokenListB = ["ME", "TOO", ",", "please", "do", "me", "next"]

lcaseListA = []

for tokA in tokenListA:
   lcaseListA.append(tokA.lower())



lcaseListB = []

for tokB in tokenListB:
   lcaseListB.append(tokA.lower())

print " ".join(lcaseListA)  # process me , please !
print " ".join(lcaseListB)  # ! ! ! ! ! ! !; why?
```

Functions: motivation

```
tokenListA = ["Process", "me", ",", "PLEASE", "!"]
tokenListB = ["ME", "TOO", ",", "please", "do", "me", "next"]

lcaseListA = []

for tokA in tokenListA:
   lcaseListA.append(tokA.lower())


lcaseListB = []

for tokB in tokenListB:
   lcaseListB.append(tokA.lower())

print " ".join(lcaseListA)
print " ".join(lcaseListB)
```

Functions: motivation

```
tokenListA = ["Process", "me", ",", "PLEASE", "!"]
tokenListB = ["ME", "TOO", ",", "please", "do", "me", "next"]

lcaseListA = []
for tokA in tokenListA:
   lcaseListA.append(tokA.lower())

lcaseListB = []
for tokB in tokenListB:
   lcaseListB.append(tokA.lower())

print " ".join(lcaseListA)
print " ".join(lcaseListB)
```

Functions: motivation

Universität Zürich[UZH]

```
tokenListA = ["Process", "me", ",", "PLEASE", "!"]
tokenListB = ["ME", "TOO", ",", "please", "do", "me", "next"]

def lcaseList(tokenList):
   result = []

   for tok in tokenList:
      result.append(tok.lower())

   return result

lcaseListA = lcaseList(tokenListA)
lcaseListB = lcaseList(tokenListB)

print " ".join(lcaseListA)
print " ".join(lcaseListB)
```

Functions: purpose

A block of code that can be called as a single instruction from within the program

- purpose
  - **enable re-using the code**
    - code shorter and more elegant
    - no copy-pasting
  - **enable structured code**
    - separate logical, independent blocks of code
    - code probably longer, but still more elegant

Functions

```
tokenListA = ["Process", "me", ",", "PLEASE", "!"]
tokenListB = ["ME", "TOO", ",", "please", "do", "me"]
```

function name

arguments/argument list

```
def lcaseList(tokenList):      function definition
    result = []                        function body

    for tok in tokenList:
        result.append(tok.lower())

    return result
```

function calls

```
lcaseListA = lcaseList(tokenListA)
lcaseListB = lcaseList(tokenListB)

print " ".join(lcaseListA)
print " ".join(lcaseListB)
```

Functions - arguments:

Also possible: no function arguments or more than one function arguments

Function Arguments:

```
def anotherFunction():
    ...

def thirdFunction(argx, argy, argz):
    ...
```

Functions - arguments:

Also possible: optional/default function arguments

Function Arguments:

```
def yetanotherFunction (argx=1, argy=[], argz=None):
    ...
```

If the function is called without arguments, the arguments get their default values.

Functions - `return` statements:

Note: a `return` statement terminates the execution of a function!

Return values:

```
def lcaseList(tokenList):
    result = []

    for tok in tokenList:
        result.append(tok.lower())

        return result

tokenListA = ["Process", "me", ",", "PLEASE", "!"]

print lcaseList(tokenListA)  # …?
```

Functions - `return` statements:

Note: a `return` statement terminates the execution of a function!

Return values:

```
def lcaseList(tokenList):
    result = []

    for tok in tokenList:
        result.append(tok.lower())

        return result

tokenListA = ["Process", "me", ",", "PLEASE", "!"]

print lcaseList(tokenListA)  # ["process"]
```

Functions - `return` statements:

`return` defines what a function returns:

Also possible: more than one `return` statement

Return values:

```
def describe_number(n):
    if n > 1000000:
        return "LARGE"
    elif n > 1000:
        return "Medium"
    else:
        return "small"
```

Note: a `return` statement terminates the execution of a function!

Functions - `return` statements:

`return` defines what a function returns:

Also possible: more than one object can be returned using a list or a tuple.

Return values:

```
def anotherFunction(a, b):
    if (...):
        return ("y", 0)
    else:
        return ("x", 1)

(c, d) = anotherFunction(a, b)

print c
print d
```

Note: a `return` statement terminates the execution of a function!

Functions - recursion:

In Python, a **recursive function** is a function which calls itself.

```
def fib(a):
    if a > 2:
        return 1
    else:
        return fib(a-1) + fib(a-2)
```

- Recursion can make definitions more elegant
- Loops and recursive functions have the same power
- Most problems that can be solved with recursion can also be solved with loops
- Recursion might have a higher complexity than a loop

Built-in functions

- Python has a wide variety of already existing functions: `http://docs.python.org/library/functions.html`
- Example:
  `sum(), len(), str(), range(), sorted()`
- Use them whenever possible!
  - partially implemented in C, in general faster
  - no re-inventing of the bicycle
  - programs more readable and understandable

Types

- Python variables have types
  - int, float
  - str
  - list
  - dict

Types

- Python variable <u>values</u> have types
  - int, float
  - str
  - list
  - dict

Types

- Python variable <u>values</u> have types
  - int, float
  - str
  - list
  - dict
  - function:

```
x = sum([1, 2, 3])
print x # …?

x = sum
print x # <built-in function sum>
```

Function Type

```
def applySomething(inputList, function):
    return function(inputList)
# functions can be used as arguments to other functions
# pass built-in / standard library / own defined function
ourInputList = [1, 1, 2, 3, 5, 8]
print applySomething(ourInputList, sum)  # 20

# pass nameless function / lambda expression
print applySomething(ourInputList,
    lambda inputList: [str(elem) + "x" for elem in inputList])
# ['1x', '1x', '2x', '3x', '5x', '8x',]



# python functions using other functions as input:
print map(str, ourInputList)  # ['1', '1', '2',...]
```

Types

- Python variable values have types
  - int, float
  - str
  - list
  - dict
  - function
- What if we need to store/pass composite information?
  - Like NLTK corpora (sentences, words, tags, …)

Composite Types

- NLTK corpora:
  - list of sentences + list of words + list of tags + …
- Storing or returning them:
  - list or tuple:

```
def text2corpus(rawText):
    …
    return (sentenceList, wordList, tagList)
```

Composite Types

- NLTK corpora:
  - list of sentences + list of words + list of tags + …
- Storing or returning them:
  - list or tuple:

```
def text2corpus(rawText):
    …
    return (sentenceList, wordList, tagList)

# bad! have to remember order and content:
corpus = text2corpus(rawInputText)
print corpus[1] # list of words
```

Composite Types

- NLTK corpora:
  - list of sentences + list of words + list of tags + …
- Storing or returning them:
  - dict:

```
def text2corpus(rawText):
    …
    return { 'sentences': sentenceList,
             'words': wordList,
             'tags': tagList }
```

Composite Types

- NLTK corpora:
  - list of sentences + list of words + list of tags + …
- Storing or returning them:
  - dict:

```
def text2corpus(rawText):
    …
    return { 'sentences': sentenceList,
             'words': wordList,
             'tags': tagList }

# better:
corpus = text2corpus(rawInputText)
print corpus["words"] # list of words

# but can do much better
```

Composite Types

- Functions meant to process a certain type of input data (like corpora):
  - save corpus / load corpus / get most frequent tokens
  - …
- Could be defined separately

```
def saveCorpus(corpus, filename): …
```

- Could be attached into the corpus variable:

```
corp = [ 'sentences': …, 'tags': …,
         'save': saveCorpus ]

corp['save'](corp, "/tmp/corpus.txt") # clumsy
```

- Could be grouped with the contents in a **class**!

Classes: overview

Classes:

- motivation
- object-oriented programming
- usage, dotted syntax
- class attributes
- class methods
- initialization method
- inheritance, overriding
- (old-style classes vs new-style classes)

Classes

```
import nltk, nltk.stem

lem = nltk.stem.WordNetLemmatizer()

sentence1 = "here are some tokens to process ."

class SentenceInfo(object):
   tokList = []
   posList = []
   lemmaList = []

info1 = SentenceInfo()

info1.tokList = sentence1.split()
info1.posList = nltk.pos_tag(info1.tokList)
info1.lemmaList = [lem.lemmatize(tok)  for tok in info1.tokList]
# dotted syntax to refer to class attributes

print info1.lemmaList
```

Classes:
Object-Oriented Programming

Universität Zürich[UZH]

Class:

A "plan of construction" for a data type. From this plan, an infinite number of objects of this data type can be created/instantiated (instances).

- **Object-Oriented Programming (OOP)**
  - OOP is programming paradigm
  - ideas are approached as objects (similar to objects in the real world)
  - objects are data structures
    - they can contain attributes (data fields/information)
    - they can have methods (functions belonging to the specific class)
  - in Python everything is an object (has a type)

Classes: Object-Oriented Programming

Universität Zürich[UZH]

```
class SentenceInfo(object):
    tokList = []
    posList = []
    lemmaList = []
```

- a class defines a type (like int/string/etc.)
  - *SentenceInfo* is a class name
- an instance of a class is an object of that type
  - *(but: beware of old-style classes in Python 2.x)*
  - instances/objects can be assigned to variables
  - a class can have as multiple instances
  - each instance is independent from other instances
- classes have attributes
  - here: variables *tokList*, *posList*, *lemmaList*
  - attributes are accessible from under an object of that class only

Classes: class attributes

```
class TmpClass(object):
    i = 5

x = TmpClass()
y = TmpClass()

print i             # NameError
print x.i           # 5
x.i = 10
print x.i   # 10
print y.i   # 5
```

Classes: class attributes

```
import nltk, nltk.stem

lem = nltk.stem.WordNetLemmatizer()

class SentenceInfo(object):
   tokList = []
   posList = []
   lemmaList = []

info1 = SentenceInfo()

info1.tokList = "here are some tokens to process ." .split()
info1.posList = nltk.pos_tag(info1.tokList)
info1.lemmaList = [lem.lemmatize(tok)  for tok in info1.tokList]

print info1.lemmaList
```

Classes: class methods

```
import nltk, nltk.stem

lem = nltk.stem.WordNetLemmatizer()

class SentenceInfo(object):
   tokList = []
   posList = []
   lemmaList = []

def createSentenceInfo(rawSentence):
   result = SentenceInfo()

   result.tokList = rawSentence.split()
   result.posList = nltk.pos_tag(result.tokList)
   result.lemmaList = [lem.lemmatize(tok)  for tok in result.tokList]

   return result

info1 = createSentenceInfo( "here are some tokens to process ." )
print info1.lemmaList[3]
```

Classes: class methods

```
import nltk, nltk.stem

lem = nltk.stem.WordNetLemmatizer()

class SentenceInfo(object):
   tokList = []
   posList = []
   lemmaList = []

   def create(self, rawSentence):
      self.tokList = rawSentence.split()
      self.posList = nltk.pos_tag(self.tokList)
      self.lemmaList = [lem.lemmatize(tok)  for tok in self.tokList]

info1 = SentenceInfo()
info1.create("here are some tokens to process ." )
print info1.lemmaList[3]
```

Classes: class methods

```
import nltk, nltk.stem

lem = nltk.stem.WordNetLemmatizer()

class SentenceInfo(object):
    tokList = []
    posList = []
    lemmaList = []

    def create(self, rawSentence):
        self.tokList = rawSentence.split()
        self.posList = nltk.pos_tag(self.tokList)
        self.lemmaList = [lem.lemmatize(tok)  for tok in self.tokList]

    def getToken(self, tokIndex):
        return (tokList[tokIndex], posList[tokIndex], lemmaList[tokIndex])

info = SentenceInfo()
info.create("here are some tokens to process ." )
print info.getToken(3)
```

Classes: class methods

Universität Zürich[UZH]

```
import nltk, nltk.stem

lem = nltk.stem.WordNetLemmatizer()

class SentenceInfo(object):
   tokList = []
   posList = []
   lemmaList = []

   def create(self, rawSentence):
      self.tokList = rawSentence.split()
      self.posList = nltk.pos_tag(self.tokList)
      self.lemmaList = [lem.lemmatize(tok)  for tok in self.tokList]
   # NB! needs to be self.tokList
   def getToken(self, tokIndex):
      return (self.tokList[tokIndex], self.posList[tokIndex],
                   self.lemmaList[tokIndex])

info = SentenceInfo()
info.create("here are some tokens to process ." )
print info.getToken(3)
```

Classes: class methods

Universität Zürich[UZH]

```
class SentenceInfo(object):
   tokList = []
   posList = []
   lemmaList = []

   def create(self, rawSentence):
      self.tokList = rawSentence.split()
      self.posList = nltk.pos_tag(self.tokList)
      self.lemmaList = [lem.lemmatize(tok)  for tok in self.tokList]

   def getToken(self, tokIndex):
      return (tokList[tokIndex], posList[tokIndex], lemmaList[tokIndex])
```

- *SentenceInfo* is the class name
- *tokList*, *posList*, *lemmaList*, *create* and *getToken*: class attributes
- *create*, *getToken*: methods/functions (class-specific functions)

Classes: syntax

class definition

```
class SentenceInfo(object):
```

class body

```
    tokList = []
    posList = []
    lemmaList = []

    def create(self, rawSentence):
        self.tokList = rawSentence.split()
        self.posList = nltk.pos_tag(self.tokList)

    def ...(self, argx, argz):
```

class name

ancestor(s)

class attributes

class methods

```
sentence_info = SentenceInfo()

sentence_info.create(raw_sent)
```

instantiation (creates a new instance of this class)

Classes: initialization method

- the __*init*__ ("initialization") method
  - is called on (= immediately after) instantiation
  - acts like a "constructor method" (but strictly speaking is not)
  - never returns a value
  - can initialize whatever necessary (e.g. starting values, additional behavior, etc.):

```
class TmpClass(object):
    i = 5

    def __init__(self, startingI):
        self.i = startingI

x = TmpClass(10)
y = TmpClass(5)
```

Classes: initialization method

```
import nltk, nltk.stem

lem = nltk.stem.WordNetLemmatizer()

class SentenceInfo:
   tokList = []
   posList = []
   lemmaList = []

   def create(self, rawSentence):
      self.tokList = rawSentence.split()
      self.posList = nltk.pos_tag(self.tokList)
      self.lemmaList = [lem.lemmatize(tok)  for tok in self.tokList]

info1 = SentenceInfo()
info1.create("here are some tokens to process ." )
info2 = SentenceInfo()
info2.create("they are already lower-cased and tokenized ." )
```

Classes: initialization method

```
import nltk, nltk.stem

lem = nltk.stem.WordNetLemmatizer()

class SentenceInfo:
   tokList = []
   posList = []
   lemmaList = []

   def __init__(self, rawSentence):
      self.tokList = rawSentence.split()
      self.posList = nltk.pos_tag(self.tokList)
      self.lemmaList = [lem.lemmatize(tok)  for tok in self.tokList]

info1 = SentenceInfo("here are some tokens to process ." )
info2 = SentenceInfo("they are already lower-cased and tokenized ." )
```

Classes: inheritance

- there is a hierarchy of classes in Python
  - object is the most basic class

- an existing class can be used as a starting point in defining a new class
- the new class
  - will include the same attributes and methods
  - may add/overwrite the attribute values and methods
- the existing class is called *parent*, *super-class or ancestor*
- the new class is called *child*, *sub-class or heir*

Classes: inheritance

Universität Zürich[UZH]

New-Style Classes vs. Old-Style Classes

```
class SentenceInfo:
    tokList = []
    posList = []
    lemmaList = []


sentence_info = SentenceInfo()
type(sentence_info)
<type 'instance'>
```

```
class SentenceInfo(object):
    tokList = []
    posList = []
    lemmaList = []


sentence_info = SentenceInfo()
type(sentence_info)
<class '__main__.SentenceInfo'>
```

- Up to Python 2.1: old-style classes
- After Python 2.1: new-style classes (but: back compatibility)
- Python 3.0: new-style classes by default (but: subclassing from object is still recommended)

New style classes: inherit from object

- some fixes, new features, ...

Classes: inheritance

```
class SentenceInfo(object):
    tokList = []
    posList = []
    lemmaList = []

    def getToken(self):
        return "something"


class MoreSentenceInfo(SentenceInfo):
    newInfo = "info"


x = MoreSentenceInfo()
print x.tokList      # []
print x.newInfo      # "info"
print x.getToken()   # "something"

print MoreSentenceInfo.__bases__   # get information about parent
```

ancestor(s)

Classes: inheritance, overriding

```
class SentenceInfo:
    someVar = 3.14
    info = "old info"

    def getToken(self):
        return self.info


class MoreSentenceInfo(SentenceInfo):
    info = "new piece of info"


x = MoreSentenceInfo()
print x.someVar       # 3.14
print x.getToken()    # "new piece of info" →
     #the getToken method of the parent class
     #is applied to the instance of the child class,
     #so the overridden variable value is taken from
     #the child class instance
```

Classes: inheritance, initialization method

```
import nltk, nltk.stem

lem = nltk.stem.WordNetLemmatizer()

class SentenceInfo:
   tokList = []
   posList = []
   lemmaList = []

   def __init__(self, rawSentence):
      self.tokList = rawSentence.split()
      self.posList = nltk.pos_tag(self.tokList)
      self.lemmaList = [lem.lemmatize(tok) for tok in self.tokList]


class MoreSentenceInfo(SentenceInfo):


   def __init__(self, raw_sentence, new_info):
     SentenceInfo.__init__(self, raw_sentence)
     self.new_info = new_info
```

- if overriding `__init__` , itialization method of super-class has to be called explicitly!

Classes: inheritance, initialization method

```
import nltk, nltk.stem

lem = nltk.stem.WordNetLemmatizer()

class SentenceInfo:
   tokList = []
   posList = []
   lemmaList = []

   def __init__(self, rawSentence):
      self.tokList = rawSentence.split()
      self.posList = nltk.pos_tag(self.tokList)
      self.lemmaList = [lem.lemmatize(tok) for tok in self.tokList]


class MoreSentenceInfo(SentenceInfo):


   def __init__(self, raw_sentence, new_info):
     super(MoreSentenceInfo, self).__init__(self, raw_sentence)
     self.new_info = new_info
```

- In python3 `super()` does not need any arguments

Modules: overview

Universität Zürich[UZH]

Modules:

- motivation
- usage, `import` command, long/short names
- module naming
- existing modules
  - file handling
  - mathematical functions
- `pip` package manager

Modules

Module:

A python file whose code can be imported and used by another python file.

- bigger programs are typically split into several files
- the executed file is called the script
- the additional files are called modules

- by default only Python's built-in types, functions, classes, etc. are available
- several modules are pre-installed (standard library)
- modules can be used via an explicit `import` command

Modules

Universität Zürich[UZH]

```
from mymodule import FreqDist

corp = ["this", "sentence", "is", "a", "sample", "sentence"]



freqDist = FreqDist(corp)
print freqDist.get("sentence")     # 0.333



########### mymodule.py:
from collections import defaultdict

class FreqDist(defaultdict):
    def __init__(self, corpus):
    # ...
```

Modules: long/short Names

```
import nltk
nltk.pos_tag(...)
```

VS

```
from nltk import pos_tag
pos_tag()
```

Module Naming

- `.py` file extension
- `[_a-zA-Z][_a-zA-Z0-9]*` file name

Imported module looked for in:

- the folder of the current script
- Python's installation path
- additionally configured paths

Module name

```
import math as m
print(m.__name__)   # math
print(__name__)     # __main__
```

- using `import` modules can be (temporarily) renamed
- *__name__* gives the actual module name
- script name is always "__main__":

```
def myfunc(x, y):
    return x + y

if __name__ == "__main__":
    print myfunc(3, 6)
```

Existing Modules

```
from nltk.probability import FreqDist

corp = ["this", "sentence", "is", "a", "sample", "sentence"]

freqDist = FreqDist(corp)
print freqDist.freq("sentence")    # 0.333
```

Existing Modules

- Interface to OS Functionality
- HTTP, FTP, E-mail
- Serializing, databases (sqlite3, pickle)
- Parallel computing
- XML
- Specific data structures
- Assisting functions
- Special theme modules
    - NLTK

Modules: file handling

`os`

- deletion
- renaming

`os.path`

- decomposition
- existence checking

Modules: file handling

```
p = "folder" + "/" + "file"
```

vs

```
p = os.path.join("folder", "file")
```

- Windows ("\") vs Unix ("/")

Modules: mathematical functions

`math`

- trigonometry
- logarithms
- special floating-poing number functions

Modules: `pip install`

pip

- Package management system for Python
  - used to install and manage packages
    - “package” = a directory of python modules
  - downloads from PyPI (“Python Package Index”): official third-party software repository for Python
  - Python 2.7.9 and later and Python 3.4 and later include pip (pip3 for Python 3) by default
  - Link to Documentation: https://pip.pypa.io/en/stable/

- On command line:
  - `pip install some-package-name`
  - `pip uninstall some-package-name`
  - `pip freeze`
  - `pip list`

Namespaces: overview

Namespaces:

- motivation
- namespaces in functions
- namespaces in classes
- namespaces in modules
- `locals()` and `globals()`
- scopes
- global names

Namespaces

Functions, classes, modules keep their own variables, functions, classes by using namespaces

Motivation

- how to solve variable/function name conflicts?
- solution: let names be valid only in a limited scope

Namespaces

- an abstract container for grouping names
- mapping from names to objects
- names in different namespaces are independent
- analogy: folders and files

Namespaces in functions

```
tokenListA = ["Process", "me", ",", "PLEASE", "!"]
tokenListB = ["ME", "TOO", ",", "please", "do", "me"]

def lcaseList(tokenList):
  result = []

  for tok in tokenList:
    result.append(tok.lower())

  return result

print result # NameError

lcaseListA = lcaseList(tokenListA)
lcaseListB = lcaseList(tokenListB)

print " ".join(lcaseListA)
print " ".join(lcaseListB)
```

Namespaces in classes

```
class TmpClass(object):
    i = 5

    def add(self, arg):
        return self.i + arg

x = TmpClass()
y = TmpClass()

print i                # NameError
print x.i              # 5
x.i = 10

print TmpClass.add(x, 2)  # 12 = 10 + 2
print TmpClass.add(x, y.i) # 15 = 10 + 5
print y.add(2)            # 7 = 5 + 2
```

Namespaces in modules

```
import nltk
nltk.pos_tag(...)
```

- nltk module imported for usage

```
from nltk import pos_tag
pos_tag()
```

- the pos_tag function imported into local namespace

Namespaces

```
def f(x):
    print(locals())

x = 20
print (globals()) # {'x': 20, 'f': <function ...>, ...}
f(10)             # {'x': 10}
print (globals()) # {'x': 20, 'f': <function ...>, ...}
```

- Python implements namespaces with dictionaries
- Namespaces are not explicitly declared
- Every module, function, class have their own namespace
- Loops and list comprehensions **do not** have their own namespace

Namespaces

```
def f(x):
     print(locals())

x = 20
print (globals()) # {'x': 20, 'f': <function ...>, ...}
f(10)             # {'x': 10}
print (globals()) # {'x': 20, 'f': <function ...>, ...}
```

- Python implements namespaces with dictionaries:

```
print x
print locals()['x']
locals()['z'] = 3
print z # 3
```

Namespaces: scopes

Universität Zürich[UZH]

- **Scope :** piece of Python code where a certain namespace can be reached directly
- In any piece of code 3 namespaces can be reached directly:
  - local scope: namespace with local names. Specific to function/class
  - global scope: namespace of the module/script
  - outer scope: namespace with built-in functions
- name search goes from local to outer
- new declarations belong to the local scope
  - (unless declared with `global`)

Namespaces: global names

```
def f(z):
    global x
    y = z
    x = z
    print(locals())

y = 20
x = 20
print(locals())   # {'x': 20, 'y': 20, 'f': <function...>}
f(10)             # {'z': 10, 'y': 10}
print(locals())   # {'x': 10, 'y': 20, 'f': <function...>}
```

Namespaces

```python
class MyClass:
    def __init__(self, i):
        self.i = i

x = MyClass(3)
print(x.i) # 3
```

- what is the difference between `self.i` and `i`?
  - `i` is in the function namespace
  - `self.i` is the namespace of `self` = object namespace
- `x` and `self` point to the same object
- `self` only valid within the function scope

To sum up

Universität Zürich[UZH]

- **classes** used to group related content and functionality
- **modules** do the same on a more general level
- **namespaces** used in Python to sort out the names of variables/classes/functions/modules/ etc. and avoid name conflicts and programmer confusion

Questions?