

Python: Short Overview and Recap

Benjamin Roth
Marina Sedinkina

CIS LMU

Data Types

Object type	Example creation
Numbers (int, float)	123, 3.14
Strings	'this class is cool'
Lists	[1, 2, [1, 2]]
Dictionaries	{'1': 'abc', '2': 'def'}
Tuples	(1, 'Test', 2)
Files	open('file.txt'), open('file.bin', 'wb')
Sets	set('a', 'b', 'c')
Others	boolean, None
Program unit types	functions, modules, classes

Variables

- store data, e.g., numbers
- content can be changed (is variable)
- have a data type
- assignment: `var_name = value`, e.g., `num = 17`

Dynamic Typing

- dynamic typing model
- types are determined automatically at runtime
- type of a variable can change
- check type with `type(var)`

Number data types

- integers, floating-point numbers, complex numbers, decimals, rationals
- Numbers support the basic mathematical operations, e.g.:

- ▶ + addition
- ▶ * , / , // multiplication, division

```
>>> 1/4
```

```
>>> 1//4
```

- ▶ ** exponentiation
- ▶ < , > , <= , >= comparison
- ▶ != , == (in)equality

String data types

- Immutable sequence of single characters

```
s1="first line\nsecond line"
```

```
s2=r"first line\nstill first line"
```

```
s3="""first line  
second line"""
```

```
s4='using different quotes'
```

- How to create the following two-line string?

```
what's up, "dude"?
```

```
-Bob
```

String operations I

```
s1 = 'the'
```

Operation	Description	Output
<code>len(s1)</code>	length of the string	3
<code>s1[0]</code>	indexing, 0-based	't'
<code>s1[-1]</code>	backwards indexing	'e'
<code>s1[0:3]</code>	slicing, extracts a substring	'the'
<code>s1[:2]</code>	slicing, extracts a substring	'th'
<code>s1 + ' sun'</code>	concatenation	'the sun'
<code>s1 * 3</code>	repetition	'thethethe'
<code>!=</code> , <code>==</code>	(in)equality	True, False

String operations II

```
s1 = 'these'
```

Operation	Description	Output
'-'.join(s1)	concatenate (delimiter: '-')	't-h-e-s-e'
s1.find('se')	finds start of substring	3
s1.replace('ese', 'at')	replace substrings	'that'
s1.split('s')	splits at string	['the', 'e']
s1.upper()	upper case	'THESE'
s1.lower()	lower case	'these'

Lists

- collection of arbitrarily typed objects
- mutable
- positionally ordered
- no fixed size
- initialization: `l = [123, 'spam', 1.23]`
- empty list: `l = []`

List operations I

```
l = [123, 'spam', 1.23]
```

Operation	Description	Output
<code>len(l)</code>	length of the list	3
<code>l[1]</code>	indexing, 0-based	'spam'
<code>l[0:2]</code>	slicing, extracts a sublist	[123, 'spam']
<code>l + [4, 5, 6]</code>	concatenation	[123, 'spam', 1.23, 4, 5, 6]
<code>l * 2</code>	repetition	[123, 'spam', 1.23, 123, 'spam', 1.23]

List operations II

```
l = [123, 'spam', 1.23]
```

Operation	Description	Output
<code>l.append('NI')</code>	append to the end	<code>[123, 'spam', 1.23, 'NI']</code>
<code>l.pop(2)</code>	remove item	<code>[123, 'spam']</code>
<code>l.insert(0, 'aa')</code>	insert item at index	<code>['aa', 123, 'spam', 1.23]</code>
<code>l.remove(123)</code>	remove given item	<code>['spam', 1.23]</code>
<code>l.reverse()</code>	reverse list (in place)	<code>[1.23, 'spam', 123]</code>
<code>l.sort()</code>	sort list (in place)	<code>[1.23, 123, 'spam']</code>

Nested lists

Let us consider the 3x3 matrix of numbers

`M = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]` . `M` is a list of 3 objects, which are in turn lists as well and can be referred to as rows.

- `M[1]` – returns the second row in the main list: `[4, 5, 6]`
- `M[1][2]` – returns the third object situated in the in the second row of the main list: `6`

Dictionaries

- Dictionaries are **mappings**, not sequences
- They represent a collection of key:value pairs
- Example:

```
d = {'food': 'Spam', 'quantity': 4, 'color': 'pink'}
```
- Efficient access (\sim constant time):
what is the value associated with a key?
- They are mutable like lists:
Key-value pairs can be added, changed, and removed
- Keys need to be immutable – why?

Dictionary operations I

```
>>> d = {'food':'Spam', 'quantity':4, 'color':'pink'}
>>> d['food']
#Fetch value of key 'food'
'Spam'
>>> d['quantity'] += 1 #Add 1 to the value of 'quantity'
>>> d
d = {'food':'Spam', 'quantity':5, 'color':'pink'}
```

Dictionary operations II

```
>>> d = {}
>>> d['name'] = 'Bob'
>>> #Create keys by assignment
>>> d['job'] = 'researcher'
>>> d['age'] = 40
>>> d
d = {'name': 'Bob', 'job': 'researcher', 'age': 40}
>>> print(d['name'])
Bob
```

Dictionary operations III

```
>>> #Alternative construction techniques:
>>> d = dict(name='Bob', age=40)
>>> d = dict([('name', 'Bob'), ('age', 40)])
>>> d = dict(zip(['name', 'age'], ['Bob', 40]))
>>> d
{'age': 40, 'name': 'Bob'}
>>> #Check membership of a key
>>> 'age' in d
True
>>> d.keys()
#Get keys
['age', 'name']
>>> d.values() #Get values
[40, 'Bob']
>>> d.items() #Get all keys and values
[('age', 40), ('name', 'Bob')]
>>> len(d)
#Number of entries
2
```


Dictionary operations IV

```
>>> d = {'name': 'Bob'}
>>> d2 = {'age': 40, 'job': 'researcher'}
>>> d.update(d2)
>>> d
{'job': 'researcher', 'age': 40, 'name': 'Bob'}
>>> d.get('job')
'researcher'
>>> d.pop('age')
40
>>> d
{'job': 'researcher', 'name': 'Bob'}
```

Tuples

- Sequences like lists but immutable like strings
- Used to represent fixed collections of items

```
>>> t = (1, 2, 3, 4) #A 4-item tuple
```

```
>>> len(t) #Length
```

```
4
```

```
>>> t + (5, 6) #Concatenation
```

```
(1, 2, 3, 4, 5, 6)
```

```
>>> t[0] #Indexing, slicing and more
```

```
1
```

```
>>> len(t)
```

```
???
```

Sets

- Mutable
- Unordered collections of **unique** and **immutable** objects
- Efficient check (\sim constant time), whether object is contained in set.

```
>>> set([1, 2, 3, 4, 3])
{1, 2, 3, 4}
>>> set('spaam')
{'a', 'p', 's', 'm'}
>>> {1, 2, 3, 4}
{1, 2, 3, 4}
>>> S = {'s', 'p', 'a', 'm'}
>>> S.add('element')
>>> S
{'a', 'p', 's', 'm', 'element'}
```

Sets

```
>>> s1 = set(['s', 'p', 'a', 'm', 'element'])
>>> 'element' in s1
True
>>> 'spam' in s1
False
>>> s2 = set('ham')
>>> s1.intersection(s2)
{'m', 'a'}
>>> s1.union(s2)
{'s', 'm', 'h', 'element', 'p', 'a'}
```

⇒ intersection and union return a new set, the original sets stay unchanged

Immutable vs. Mutable

- Immutable:
 - ▶ numbers
 - ▶ strings
 - ▶ tuples
- Mutable:
 - ▶ lists
 - ▶ dictionaries
 - ▶ sets
 - ▶ newly coded objects

Control flow: if-statements

```
>>> x = 'killer rabbit'
... if x == 'roger':
...     print('shave and a haircut')
... elif x == 'bugs':
...     print('whats up?')
... else:
...     print('run away!')
run away!
```

Note!

The `elif` statement is the equivalent of `else if` in Java or `elsif` in Perl.

Control flow: While loops

```
>>> while True:
...     print('Type Ctrl-C to stop me!')
...
>>> x = 'spam'
... while x: #while x is not empty
...     print(x)
...     x = x[1:]
...
spam
pam
am
m
```

⇒ `x[len(x):len(x)]` returns the empty string.

Control flow: For loops

The for loop is a generic iterator in Python: it can step through the items in any ordered sequence or other iterable objects (strings, lists, tuples, and other built-in iterables, as well as new user-defined iterables).

```
l = [1, 2, 3, 4]
```

```
for i in l:  
    print(i)
```

```
for i in range(0, 5):  
    print(i)
```


Files: Read file line by line

```
file_name = '/path/to/file.txt'
with open(file_name, mode='r') as f:
    for line in f.readlines():
        # Lines still contain line-break.
        # Print without newline:
        print(line, end='')
```

How to remove trailing new line?

Files: Write file line by line

```
file_name = '/path/to/file.txt'
lines = ['line1', 'second line', 'another line', 'last one']

with open(file_name, mode='w') as f:
    for line in lines:
        f.write(line + '\n')
```

Functions

- A function is a device that groups a set of statements so they can be run more than once in a program
- Why use functions?
 - ▶ Maximizing code reuse and minimizing redundancy
 - ▶ Procedural decomposition

Defining functions

```
def name(arg1, arg2, ..., argN):  
    statements
```

```
def name(arg1, arg2, ..., argN):  
    ...  
    return value
```

```
>>> def times(x, y):  
...     return x*y  
...  
>>> times(2, 5)  
10
```

Function objects

```
def func():  
    #Create function object  
    ...  
func() # Call object.  
func.attr = value # Attach attributes.
```

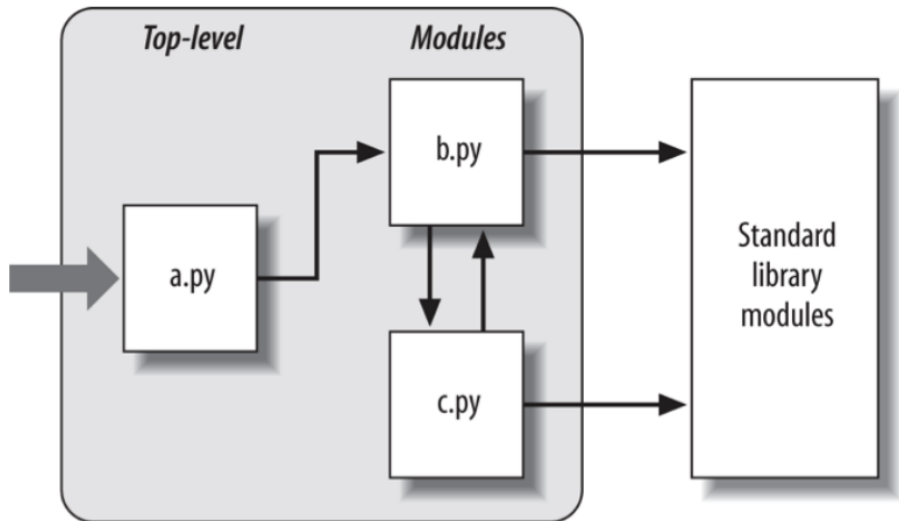
Module

- Packaging of program code and data for reuse
- Provides self contained namespaces that avoid variable name clashes across programs
- The names that live in a module are called its attributes
- one Python file \sim one module
- Some modules provide access to functionality written in external languages such C++ or Java. (wrappers)

Module

- `import` – Lets a client (importer) fetch a module as a whole
- `from` – Allows clients to fetch particular names from a module

Module imports



Regular expressions

- A regular expression is an algebraic formula (a pattern) that represents a set of strings.
- Can be used to search for strings that match the pattern.

Regex	Strings
a	a
ab	ab
a*	€, a, aa, aaa, ...
a*b*	€, a, b, aa, ab, bb, ..., aaaaaab, ...

Regular Expressions

What can you match with the following regular expressions?

- ❶ `'^[Tt]the\b .*'`
- ❷ `'[:;]-?[\|opPD\)\(]'`
- ❸ `'<.*?>'`
- ❹ `'\d +\ -year\ -old'`

- Documentation:

<https://docs.python.org/3/library/re.html>

- Test your regex online: <https://pythex.org/>

Regular Expressions

- To use Regular Expressions in Python, import the module `re`
- Then, there are two basic ways that you can use to match patterns:
 - ▶ `re.match()`:
Finds match of pattern at the beginning of a string
 - ▶ `re.search()`:
Finds match of pattern anywhere in a string `re.match()`
- Both return a *match* object, that stores more information about the match, and `None` when there is no match.

Regular Expressions

```
import re
wordlist = ['farmhouse', 'greenhouse', 'guesthouse']
for w in wordlist:
    if re.match('(g.*?)(?=house)', w):
        print(w)
```

```
match = re.search(pattern, string)
if match:
    match_str = match.group(0)
```

Compiling regular expressions

If the same regular expression is used repeatedly (in a loop), it is more efficient to compile it outside of the loop.

```
import re
wordlist = ['farmhouse', 'greenhouse', 'guesthouse']
regex = re.compile('(g.*?)(?=house)')
for w in wordlist:
    if regex.match(w):
        print(w)
```

Python classes

```
class Classifier:
    def __init__(self, lambda1, lambda2):
        self.l1 = lambda1
        self.l2 = lambda2

    def train(self, data):
        ....

    def test(self, data):
        ....

if __name__ == '__main__':
    data = 'This is training data'
    testdata = 'This is test data'
    lambda1 = 0.002
    lambda2 = 0.0005
    model = Classifier(lambda1, lambda2)
    model.train(data)
    model.test(testdata)
```

Summary

- Data types: numbers, strings, tuples, lists, dictionaries
- Mutable / Immutable
- If-statement, while-loop, for-loop
- Reading / writing from files
- Functions
- Importing modules
- Regular expressions
- **Any questions?**