Python

Strings and mutability

Prof. Dr. Thomas Kopinski

Python - strings

Strings:

- contain: digits, spaces, special characters, letters
- come in quotation marks
- can be concatenated

```
3  s1 = 'Python is a programming language and an animal'
4
5  s2 = "this is a snake and it is " + str(5) + " metres long"
6
7  s3 = 'that is a ' + 'very '*3 + 'long animal'
8
9  print (s1)
10 print (s2)
11 print (s3)
```

[strings.py]

- `str()` converts to string
- mathematical operations on strings are possible

```
bash-4.4$ python strings.py
Python is a programming language and an animal
this is a snake and it is 5 metres long
that is a very very very long animal
```

[terminal output]

Python - strings

Strings:

- `print ()` - prints result to output

not ok! leads to error

```
13  print ( 'The data file is ' + 5 + ' MB large')
14  print ( 'The data file is ' + str(5) + ' MB large')
```

- strings and numericals cannot be mixed - need for conversion

```
Traceback (most recent call last):
  File "strings.py", line 13, in <module>
    print ( 'The data file is ' + 5 + ' MB large')
TypeError: can only concatenate str (not "int") to str
```

Input:

```
16  user_input = input("Enter a message... ")
17  print('User entered: ' + user_input)
```

Python - strings

Strings:

- `print ()` - prints result to output

```
s4 = 'This'
s5 = 'This'
s6 = 'this'

print(s4 == s5)
print(s4 == s6)
print(s4 is s5)
print(s4 is s4)
```

```
True
False
True
True
```

Python | `is vs ==`

- `is` - returns `True` if two variables point to the same object
- `==` - returns `True` if two objects referred to by the variables are equal
-

```
a = [1,2,3,4,5]
b = a
print('b is a? ' + str(b is a))
print('b is a? ' + str(b == a))
```

```
b is a? True
b is a? True
```

```
b = a[:]
print('b is a? ' + str(b is a))
print('b is a? ' + str(b == a))
```

```
b is a? False
b is a? True
```

Python | caching of small objects

```
34  print('caching example:')
35  print(10000 == 10**4)
36  print(10000 is 10**4)
37
38  print('a' is 'a')
39  print('aa' is 'a'*2)
40  x = 'a'
41  print('aa' is x * 2)
42  print('aa' is sys.intern(x * 2))
```

```
caching example:
True
True
True
True
False
True
```

- `is` - use ***is*** for reference equality
- `==` - use **==** for value equality

Python | sys.intern

```
string = 'myString'
print(sys.intern(string))
```

```
myString
```

- from the Python documentation:

> Enter string in the table of “interned” strings and return the interned string – which is string itself or a copy. Interning strings is useful to gain a little performance on dictionary lookup – if the keys in a dictionary are interned, and the lookup key is interned, the key comparisons (after hashing) can be done by a pointer compare instead of a string compare. Normally, the names used in Python programs are automatically interned, and the dictionaries used to hold module, class or instance attributes have interned keys.
>
> Interned strings are not immortal; you must keep a reference to the return value of intern() around to benefit from it.

- what it means:
  - use sys.intern fpr performance optimization
  - sys.intern maintains a table of interned strings - strings passed to the function are looked up and
    - either includes it in the table, if it does not exist there yet, and returns it from the table
    - or it exists in the table and therefore is returned directly from it

Python | sys.intern

```
stringA = sys.intern('myString')
stringB = sys.intern('myString')
print(stringA)
print(stringB)
```

```
myString
myString
```

- in the example above, the `stringB` object now holds the same string object as `stringA` - why?

```
print ( '\nresult of intern experiment:')
stringA = sys.intern('myString'*1000)
stringB = sys.intern('myString'*1000)

print(stringA is stringB)

stringC = 'myNewString'*1000
stringD = 'myNewString'*1000

print(stringD is stringC)
```

```
True
False
```

- by interning strings, we save memory
- comparison of strings is now handled via memory addresses instead of values —> very efficient

Python | strings

- indexing in strings with square brackets
- comparison of strings is possible
- `len()` returns the length of a string as `int`

```
stringA = 'abc'
stringB = 'abcde'

print(stringA <= stringB)
print(len(stringB))

print(stringA == stringB[:-2])
```

```
True
5
True
```

- slicing of strings is possible using the syntax `[start:stop:step]`
- default value of `step` is 1
- numbers can be left out and colons used instead

```
stringC = 'abcdefghijkl'
print(stringC[::])
print(stringC[::2])
```

```
abcdefghijkl
acegik
```

Python | strings

- strings cannot be changed, i.e. are immutable objects

```
stringA[0] = 'A'
```

```
Traceback (most recent call last):
  File "strings_2.py", line 14, in <module>
    stringA[0] = 'A'
TypeError: 'str' object does not support item assignment
```

Python | mutable vs. immutable objects

Containers:

- containers in Python can hold references to other objects
- containers are
  - lists
  - tuples
  - dictionaries

Python | mutable vs. immutable objects

- in Python, everything is an object
- objects are assigned object IDs
- identity:
  - this is the address of an object and it never changes
  - `is` operator compares the identity
  - `id()` returns the identity
- type:
  - types of objects are defined at runtime
  - determines which functions are possible on an object (e.g. `len()` on a string)
  - type() returns an object's type
  - types are unchangeable
- value:
  - **mutable** objects can be changed - i.e. their value can change
  - **immutable** objects cannot be changed - i.e. their value cannot change
  - this mutability is determined by the type

Python | mutable vs. immutable objects

mutability vs. immutability

- mutability means, we can change the content of an object without changing its identity
- objects like e.g. floats, strings are immutable, meaning we cannot change the object without changing its identity
- to understand this, we need the function `id()`

```
print('\nidentity example')
#identity example with a float
f = 7
print(id(f))
f = 8
print(id(f))
```

```
identity example
4335031632
4335031664
```

- in the example above we see that we can assign a new value to f
- however, the identity of the object also changes

Python | mutable vs. immutable objects

mutability vs. immutability

- strings can be extended, however their id also changes!

```
d1 =  'data'
print(id(d1))
d1 += ' science'
print(id(d1))

print(d1)
```

```
140371611550384
140371074239344
data science
```

```
l1 = [1,2,3]
print(id(d1))
l1 += [4,5,6]
print(id(d1))
print (l1)
```

```
140403689704304
140403689704304
[1, 2, 3, 4, 5, 6]
```

Python | mutable vs. immutable objects

mutability vs. immutability

	mutable types	immutable types
numbers		`int()`, `float()`, `complex()`, `decimal()`
sequences	`list()`, `bytearray()`	`str()`, `tuple()`, `frozenset()`, `bytes()`, `range()`
set type	`set()`	
mapping type	`dict()`	
other	`class`, `class instance`	`bool()`

Python | mutable vs. immutable objects

- in this example we see the assignment of one list to another
- if the value of the latter is changed, so is the value of the former

```
l2 = [1,2,3]
l3 = l2

l3 += [4,5]
print(l2)
print(l3)
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

- as we can see, the identity of objects l2 and l2 in this example are the same, i.e. they point to the same address in memory

```
l2 = [1,2,3]
l3 = l2

l3 += [4,5]
print(l2)
print(l3)

print(id(l2))
print(id(l3))
```

```
140460597414880
140460597414880
```

Python | mutable vs. immutable objects

- assigning a variable to a mutable object is possible
- in consequence the values of both changes, as the have the same identity, when the value of either one is changed

- the same thing does not hold for immutable objects

```
print('\nimmutable objects:')
s1 = 'this '
s2 = s1

print(s2 is s1)
print(id(s1))
print(id(s2))
```

```
immutable objects:
True
140465832217456
140465832217456
```

```
s1 += 'is Python'

print('\nafter change')
print(s1)
print(s2)
print(s2 is s1)
print(id(s1))
print(id(s2))
```

```
after change
this is Python
this
False
140465832217712
140465832217456
```

Python | mutable vs. immutable objects

- updating values of immutable objects creates a new object!
- there is a different behaviour when dealing with container objects:

```
print('\ncontainer example')
data = [1,2,3,4,5]
tup = ('someString', data)

print(data)
print(id(tup))

data[3] = 'change'

print(data)
print(id(tup))
```

```
container example
[1, 2, 3, 4, 5]
140400467641680
[1, 2, 3, 'change', 5]
140400467641680
```

- the identity of the tuple did not change, although we changed a value of a container it was holding!

Python | mutable vs. immutable objects

Special case:

- updating values of immutable objects in an immutable container

```
print('\nspecial case\n')
str1 = 'myString'
int1 = 25
tup1 = ('a','b','c')
tup2 = (str1, int1, tup1)

print('id str1:' + str(id(str1)))
print('id int1:' + str(id(int1)))
print('id tup1:' + str(id(tup1)))
print(tup2)
print('id tup2:' + str(id(tup2)))
```

```
special case

id str1:140452007464240
id int1:4427212688
id tup1:140452007280144
('myString', 25, ('a', 'b', 'c'))
id tup2:140452142349040
```

```
str1 += ' has changed'
int1 = 500
tup1 += ('d', 'e')

print('\nafter change:\n')

print('id str1:' + str(id(str1)))
print('id int1:' + str(id(int1)))
print('id tup1:' + str(id(tup1)))
print(tup2)
print('id tup2:' + str(id(tup2)))
```

```
after change:

id str1:140452007485152
id int1:140452142169520
id tup1:140452410780368
('myString', 25, ('a', 'b', 'c'))
id tup2:140452142349040
```

- identity of the container remains!
- the change of the values of immutable objects creates new objects instead!

Python | summary

- data is stored in objects
- relations between objects can be established
- objects have identities, types and values
- identities of objects
  - are established on creation
  - can be checked with `id()`
  - refer to addresses in memory
  - can be compared using `is` operator
- types of objects define their value and their operations
- *mutable* objects: their value can change
- *immutable* objects: value is unchangeable
- *mutability of a container* refers to the identity of this container