KOLT Python

Containers, Aliasing & Mutability

Ahmet Uysal

Monday 18th March, 2019

Agenda

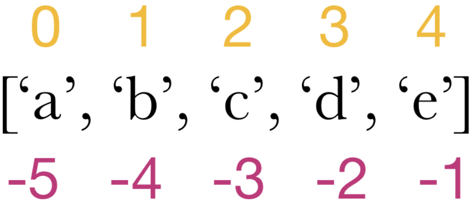
1. Recap
2. Lists
3. Mutability
4. Aliasing & Cloning
5. Tuples
6. Sets
7. Dictionaries

Lists

- Group values together. `my_values = [1, 'a', None]`
- You can think of each element as a variable, accessed by **indexing**
- You can do everything you do to variables to list elements:
  - Assign new values: `my_values[0] = 3`
  - Use shorthand assignment operators: `my_values[1] += 'bc'`
  - Learn their type: `type(my_values[2]) # => <class 'NoneType'>`
  - Change their type: `my_values[2] = True`
  - Compare their value: `if my_values[0] == my_values[1]: ...`
- What happens when we call `my_values[3] = 3`? `# =>` **`IndexError`**

Indexing

Access elements at a particular index



```
x = [1, 2, 'a', 'hello']
x[0] # => 1
x[1] # => 2
x[2] # => 'a'
x[3] # => 'hello'
x[-1] # => 'hello'
x[-2] # => 'a'
x[-3] # => 2
```

Slicing

Access collection of elements by specifying `[start:stop:step]`
Gives a list, even when number of elements is not bigger than 1.

```
numbers = [0, 1, 2, 3, 4, 5]
"""
ASCII art analogy :)
 +---+---+---+---+---+---+
 | 0 | 1 | 2 | 3 | 4 | 5 | => Indices
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6 => Borders
-6  -5  -4  -3  -2  -1
"""
```

```
numbers[0::2]   # => [0, 2, 4]
numbers[:]   # => [0, 1, 2, 3, 4, 5]
numbers[1:]   # => [1, 2, 3, 4, 5]
numbers[-2:]   # => [4, 5]
numbers[1:4]   # => [1, 2, 3]
numbers[1:1]   # => []
numbers[-99:99]   # => [0, 1, 2, 3, 4, 5]
numbers[::-1]   # => [5, 4, 3, 2, 1, 0]
numbers[::-2]   # => [5, 3, 1]
```

Slices with `step` = 1 are called **Basic Slice**.
Slices with `step != 1` are called **Extended Slice**.

Strings

Special kind of **lists**! `name = 'Ahmet'`
You can do:

- Indexing: `name[2]` ⇒ 'm'
- Slicing: `name[::-1]` ⇒ 'temhA'
- Search by `in` operator: `'hm'in name` ⇒ `True`

You can not do:

- String mutation: name[2]='H' ⇒ **`TypeError`**

Special functions about strings: `str.isnumeric()`, `str.capitalize()`, `str.format(...)`, `str.find() ...`

Loops

Do something for many elements or based on a condition.

```
while <condition>:
    <expression>
    <expression>
    ...
```

```
for <item> in <iterable>:
    <expression>
    <expression>
    ...
```

Similar to simple if blocks, but runs again and again until condition check fails.

Iterable: collection of **ordered** elements.
What is next after this item?

For Loops

What is next after this item?
numbers[1] is after numbers[0] $\neq$ **numbers[1]** $>$ **numbers[0]**
Examples of iterables: lists, strings, ranges

Ranges

`range(start, stop, step)`: creates a sequence of integers from start (inclusive) to stop (exclusive) by step.
Can be **indexed** and **sliced**
`len()` and `in` operator can be used

Break, Continue

Break terminates the closest for or while loop

```
for i in range(0, 5):
    if i % 2 == 1:
        break
    print(i)
```

```
x = 1
while x < 100:
    x *= 2
    if (x+1) % 3 == 0:
        break
    print(x)
```

Continue continues with the next iteration of the loop

```
for i in range(0, 5):
    if i % 2 == 1:
        continue
    print(i)
```

```
x = 1
while x < 100:
    x *= 2
    if (x+1) % 3 == 0:
        continue
    print(x)
```

List Mutation

`list.append(x)`: Append x to end of the sequence
`list.insert(i, x)`: Insert x to index i
`list.pop(i=-1)`: Remove and return element at index i
`list.remove(x)`: Remove first occurrence of x
`list.extend(iterable)`: Add all elements in iterable to end of list
`list[i] = new_value`: Update value of index i with new value
`list[basic_slice] = iterable`: Change elements in basic slice with elements in iterable, sizes can be different: `numbers[:] = []`
`list[extended_slice] = iterable`: Change elements in extended slice with elements in iterable 1-1, sizes must be equal.

Some Other List Operations

`in` operator: Check whether an element is in list. `3 in numbers` ⇒ `True`

`len(list)`: Returns the length of list(and other collections).

`list.index(value, start=0, stop=len(list))`: Return first index of value.

`list.count(value)`: Count number of occurrences of value in list.

`list.reverse()`: Reverse the list (in-place)

`list.sort()`: Sort list elements (in-place)

For more, type `help(list)` in your interactive interpreter.

Mutability

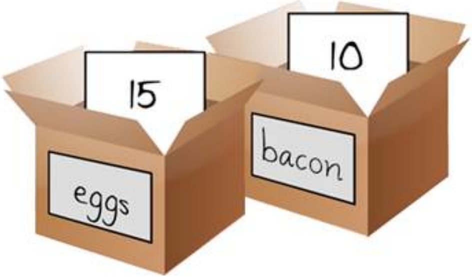
Immutable:

An `object` with a fixed value. Immutable objects include **numbers**, **strings** and **tuples**. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant **hash value** is needed, for example as a **key** in a `dictionary`.

```
a = 5
a = 10
a += 3
```

Python Data Model

How did we represent data in Python? **Variables!**
How do they work? Do they store the data themselves?



Box Analogy

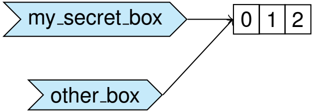
```
my_fav_number = 13
other_number = my_fav_number
other_number += 3
print(my_fav_number)   # => 13
```

```
my_secret_box = [0, 1, 2]
other_box = my_secret_box
other_box.remove(2)
print(my_secret_box)   # => [0, 1]
```

Did we just changed inside of a closed box? Box analogy **does not** work!

Python Data Model

```
my_secret_box = [0, 1, 2]
other_box = my_secret_box
other_box.remove(2)
print(my_secret_box)
```



Variables are more like **labels** pointing to **values**!

Assignment links **variables** to **values**!

Mutability

Immutable:

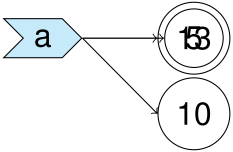
An `object` with a fixed value. Immutable objects include **numbers**, **strings** and **tuples**. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant **hash value** is needed, for example as a **key** in a `dictionary`.

```
a = 5
a = 10
a += 3
```

Object

Everything is an object in Python. Even though variables **do not** have `types`, each object has a **fixed** `type`.
↪ Values at the right side of our label analogy are objects!

```
a = 5
a = 10
a += 3
print(a)
```



Object

Each object has an `identity`, this value can be obtained by using **`id()`** function.

`==` operator compares values, **`is`** operator compares identities.

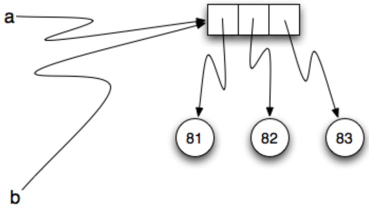
```
a = 1000
b = 1000
a == b   # => True
a is b   # => False
```

Almost always use **`==`** to compare values!

Aliasing & Cloning

- More than one variables can refer to **same object**!
- What if we want to clone/copy instead of aliasing?
- For lists, `list.copy()` ⇒ returns a shallow copy of the list.
- Shallow: only copy the references, not inner values.
- `>>> import copy`

`copy.copy(x)`: shallow copy, `copy.deepcopy(x)`: deepcopy



Tuples

- **Immutable** sequence(ordered) of elements.
- Similar to `lists`, you can use **indexing**, **slicing**, and iterate over using `for` loops.
- Elements cannot be added/removed/changed once the tuple is created.
- How to create tuples? **`my_tuple = (1, [1, 2], 'a')`**
- **`len(my_tuple)`** ⇒ 3
- **`my_tuple.append(3)`** ⇒ **`AttributeError:`** `'tuple' object has no attribute 'append'`

Tuples

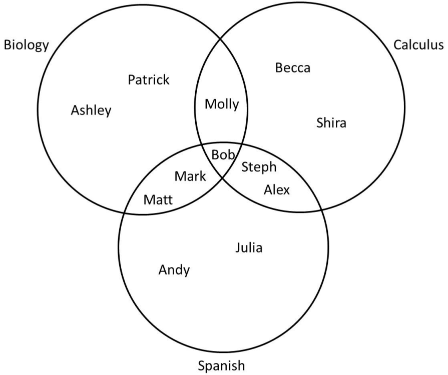
`() / tuple()`: empty tuple, `(3)`: `int` 3, `(3,)`: `tuple` containing 3

```
my_list = [1, 2, 3]
my_tuple = ('a', my_list)
my_list.append(4)
print(my_tuple)
my_list += [5, 6, 7]  # my_list.extend(...)
print(my_tuple)
my_tuple += (1, 2)  # my_tuple = my_tuple + (1, 2)
print(my_tuple)
```

Sets

- **Unordered** sequence of **unique** elements.
- **Cannot** use **indexing/slicing**, **can** iterate with `for` loops.
- **Mutable**, `add(element)`, `remove(element)` methods.
- Python also has **immutable** sets: `frozenset`
- How to create sets? `my_set = {1, 2, 3, 4, 2}`
- How to create empty sets? `set()` ({ } is reserved for `dict`)
- Can compute set operations: **union**, **intersection**, **difference**, **symmetric difference**.

Sets


Biology
Calculus
Spanish
Patrick
Ashley
Molly
Becca
Shira
Bob
Mark
Matt
Steph
Alex
Julia
Andy

Sets

```
biology = {'Ashley', 'Patrick', 'Molly', 'Bob',
           'Mark', 'Matt'}
calculus = {'Becca', 'Shira', 'Alex', 'Molly', 'Bob', 'Steph'}
spanish = {'Matt', 'Mark', 'Bob', 'Alex', 'Steph', 'Julia', 'Andy'}
# intersection &
print(biology.intersection(calculus))  # => {'Molly', 'Bob'}
print(calculus & spanish)  # => {'Bob', 'Alex', 'Steph'}
# union |
print(biology.union(calculus))  # => all names except andy and julia
print(calculus | spanish | biology)  # => all names
# difference -
print((biology - calculus).intersection(spanish))  # => {'Mark', 'Matt'}
# symmetric_difference ^
print(biology.symmetric_difference(spanish))
# => {'Molly', 'Julia', 'Ashley', 'Alex', 'Steph', 'Andy', 'Patrick'}
```

Dictionaries

- Collection of **key**—**value** pairs.
- **Cannot** use **indexing/slicing**, **can** iterate with `for` loops.
- In general, they are not **ordered**.
- However, in Python 3.7 pairs are guaranteed to be in insertion order.
- In other words, we will get pairs in insertion order if we loop over the `dict`.
- How to create dictionaries? `{ }`/`dict()`: empty dictionary
- `d = {'one': 1, 'two': 2, 'three': 3, 'four': 4}`
- How to access values? `print(d['one'])` # ⇒ 1

Dictionaries

```python
d = {'x': 1, 'y': 2, 'z': 3}
for key, value in d.items():
    print(f'value {value} is associated with key: {key}')

for key in d:
    print(f'value {d[key]} is associated with key: {key}')

# Add new pairs
d['a'] = 15
# Change value of key
d['x'] = 1
# Remove pairs
y_value = d.pop('y')
```