The UK's European university

University of Kent

COMP8270 / PROGRAMMING FOR ARTIFICIAL INTELLIGENCE

Fernando Otero
febo@kent.ac.uk
cs.kent.ac.uk/people/staff/febo

overview:

1. Data structures
   - Lists
   - Dictionaries
   - Sets
   - Tuples

2. Loops + Data structures

overview:

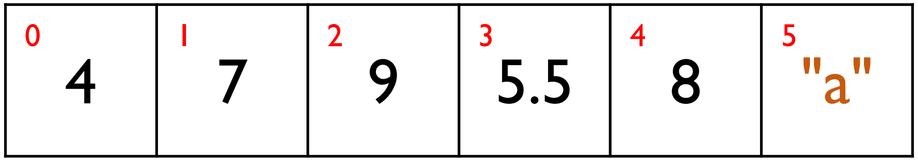
1. Data structures
   - Lists
   - Dictionaries
   - Sets
   - Tuples

2. Loops + Data structures

Lists:

- Flexible **ordered** collection of (any) object type

- Main characteristics:
  - Heterogeneous: can store multiple data types in the same list
  - Variable-length: much like an `ArrayList` in Java



List creation (1):

```
a_list = list()  # using the list() type function

a_list = []      # using []'s
```

- In both cases, `a_list` is an empty list

List creation (2):

- Surrounding values between `[]'s` creates a list

```
b_list = [5]         # 1 element int
b_list = ['a']       # 1 element string
b_list = [1, 2, 'a'] # 2 elements int, 1 string
```

- You can also use the constructor with any **iterable**

```
b_list = list("Fernando")
# ['F', 'e', 'r', 'n', 'a', 'n', 'd', 'o']
```

Adding elements to a list:

- `append()`: adds an element to the end of the list

```
b_list = ["Two"]
b_list.append("Three")
# ["Two", "Three"]
```

- `insert()`: inserts an element at a specific position
  - if index < 0, inserts at index 0
  - if index > length, inserts at the end of the list

```
b_list.insert(0, "One")
# ["One", "Two", "Three"]
```

Removing elements:

- `pop()`: removes an element at specific index
  - returns the removed element
  - if no index specified, removes the last element

```
a_list = ["One", "Two", "Three"]
a_list.pop(1)
# ["One", "Three"]
```

- `remove()`: removes the specified value from the list

```
a_list.remove("One")
# ["Three"]
```

Combining lists:

- Adding lists with +
  - creates a **new list**

```
a_list = ["One", "Two", "Three"]
b_list = ["Four"]
combined = a_list + b_list
# ["One", "Two", "Three", "Four"]
```

- `extend()`: **appends** multiple elements to a list

```
a_list.extend(b_list)
# a_list = ["One", "Two", "Three", "Four"]
```

Dictionaries (`dict`):

- Collection of *key-value* pairs – a hash map
  - mapping between *key* → *value*
  - keys are **unique**
  - key-value pairs are not in a particular order

key	value
eggs	3
muffin	5
toast	4
ham	1

dict creation:

```
empty_dict = dict()  # using the dict() type function

empty_dict = {}      # using {}'s
```

- or using colons to separate keys and values:

```
d1 = {"eggs" : 3, "muffin" : 5, "toast" : 4, "ham" : 1}
```

Adding and retrieving values:

```
# adds a new (key, value) pair
d1["a"] = "some value"
d1[7] = "an integer"

# retrieves a value from the dict
print(d1["a"])
# prints "some value"
```

- Trying to retrieve a value from a key that is not in the `dict` throws an error

Updating and removing:

```
# updates item with key "a"
d1["a"] = "some other value"

# removes item with key "a"
d1.pop("a")
```

Sets:

- Set is an **unordered** collection of **unique** elements
    - Like keys of a `dict`

- Main characteristic: do not allow duplicated values!

4	7	"b"	5	8	7.2

Set creation:

```
s1 = set()        # using the set() type function

s2 = {1, 2, 3, 4} # using {}’s but the set cannot be empty
```

- The `{}` notation is the same as the `dict`, but a set contains a sequence of values
    - Empty `{}`’s creates a `dict`

Adding / Removing elements:

- `add()`: adds an element to the set
  - No guarantee on their ordering

```
s1 = set()
s1.add("first")
# {"first"}
```

- `remove()`: removes an element
  - if the value is not present, generates an error

```
s1.remove("first")
# {}
```

Set operations:

- `union()`: distinct values occurring in **either** set
  - returns a new set

```
s1 = {1, 2, 3}
s2 = {3, 4, 5}
s1.union(s2)
# {1, 2, 3, 4, 5}
```

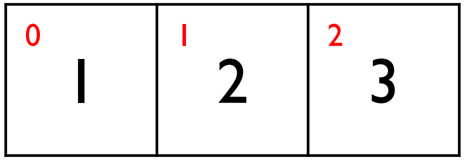
- `intersection()`: values occurring in both sets

```
s1.intersection(s2)
# {3}
```

Tuples:

- Simple group of object

- Main characteristics:
  - Ordered, indexed sequence (similar to lists)
  - Immutable: values cannot be changed



0	1	2
1	2	3

Tuple creation:

```
t1 = ()               # an empty tuple

t2 = (1, 2)           # tuple with 2 elements

t3 = tuple([3, 4, 5]) # tuple from a list
```

Tuple creation:

```
t3 = tuple(["foo", [1, 2], True]) # tuple from a list
# ("foo", [1, 2], True)

t3[1].append(3)
# ("foo", [1, 2, 3], True)
```

- Remember: values cannot change after creation, unless the object in the tuple is mutable

Unpacking tuples:

- Assign tuple values to variables:

```
t1 = (1, 2, 3)
a, b, c = t1
# a = 1
# b = 2
# c = 3
```

- Swap values of variables:

```
a, b = 1, 2      # a = 1, b = 2
b, a = a, b
# a = 2, b = 1
```

Length:

- `len()`: returns the number of elements
  - works for lists, sets, dictionaries and tuples

```
a_list = [1, 2, 3, 4]
len(a_list)
# 4
```

- Things that you normally do not do:

```
index = 0

while index < len(a_list):
    print(a_list[index])
    index += 1
```

overview:

1. Data structures
   - Lists
   - Dictionaries
   - Sets
   - Tuples

2. Loops + Data structures

Loops:

- All data structures support iteration:
  - they implement the `__iter__()` method

- Lists, sets and tuples:

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

for value in numbers:
    print(value)
```

Loops:

- Sometimes you would like to use the index of the element

- `enumerate()`: returns a sequence of `(i, value)` tuples

```python
numbers  = [1, 2, 3, 4, 5]


for i, value in enumerate(numbers):
    print(value, "at index", i)
```

Loops:

- Dictionaries:

```
colours  = {"blue" : 3, "red" : 5}
# iterates over the keys
for key in colours:
    print(colours[key])   # prints the value

# iterates over the values
for value in colours.values():
    print(value)          # prints the value
```

Loops:

- Dictionaries:

```
# iterates over the keys and values
for key, value in colours.items():
    print(value, "with key", key) # prints key and value
```

Next lecture:

- Comprehension and slicing