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. Comprehensions

2. Slicing

overview:

1. Comprehensions

2. Slicing

Comprehensions:

- One of the most-loved features of Python

- Allows you to concisely form a new list by:
  - filtering the elements of a collection
  - transforming the elements passing the filter

- … in one concise expression!

- Easier to write and read

List comprehension:

- They take the basic form:

  [*expr* for **val** in **collection** if *condition*]

- This is equivalent to the following for loop:

```
result = []
for val in collection:
    if condition:
        result.append(expr)
```

Example:

- Given a list of strings, filter out strings with length 2 or less and also convert them to uppercase:

```
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
# ...
result = ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

Option 1:

- Use a for + if statement:

```
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
result = []


for s in strings:
    if len(s) > 2:
        result.append(s.upper())


print(result)
```

Option 2:

- List comprehension:

```
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
print([s.upper() for s in strings if len(s) > 2])
```

- quite shorter! ☺

Another Example:

- Given a list of strings, ~~filter out strings with length 2 or less and also~~ convert them to uppercase:

```
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
print([s.upper() for s in strings])
```

- The filter (if *condition*) can be omitted, leaving only the *expr*

Set comprehension:

- Looks like the equivalent list comprehension, but uses curly braces instead of squared:

{*expr* for **val** in **collection** if *condition*}

Example (1):

- Given a list of strings, create a set containing the unique lengths of the strings contained in the collection:

```
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
# ...
unique_lengths = {1, 2, 3, 4, 6}
```

Example (2):

- Given a list of strings, create a set containing the unique lengths of the strings contained in the collection:

```
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
unique_lengths = {len(s) for s in strings}
```

Dict comprehension:

- Similar, but need to specify both key and value expressions:

{*key-expr* : *value-expr* for **item** in **collection** if *condition*}

- As you probably noticed, the result of the comprehension is a collection of the desired type

Example (1):

- Using the same list of strings, create a dictionary to allow the lookup of the index in the list of each string:

```
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
# ...
mapping = {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

Example (2):

- Using the same list of strings, create a dictionary to allow the lookup of the index in the list of each string:

```
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
mapping = {value : index for index, value in enumerate(strings)}
```

Nested comprehensions:

- Useful when you are dealing with a collection of tuples or collection of collections:

```
# list of lists
all_names = [['John', 'Emily', 'Michael', 'Mary', 'Steven'],
             ['Maria', 'Juan', 'Javier', 'Natalia', 'Pilar']]

# list of tuples
some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
```

Example (1):

- Filter the names to create a single list containing all names with two or more e’s in them:

```
names_of_interest = []
for names in all_names:
    # names is a list
    enough_es = [name for name in names if name.count('e') >= 2]
    names_of_interest.extend(enough_es)
```

- This uses comprehension as we have seen so far

Example (2):

- Filter the names to create a single list containing all names with two or more e's in them:

```python
names_of_interest = [name for names in all_names for name in names
                     if name.count('e') >= 2]
```

Another Example (1):

- “Flatten” a list of tuples of integers into a simple list of integers:

```
some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

# ...

flattened = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Another Example (2):

- “Flatten” a list of tuples of integers into a simple list of integers:

```
some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
flattened = [x for tup in some_tuples for x in tup]
```

overview:

1. **Comprehensions**

2. **Slicing**

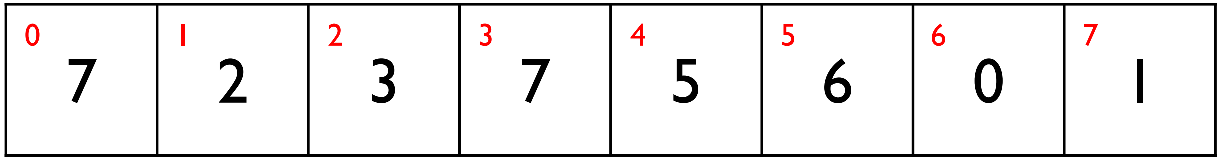
Slicing:

- Easy way to select section of sequence types

- Simple notation: `[start:stop]`
  - stop index **not** included

```
seq = [7, 2, 3, 7, 5, 6, 0, 1]


print(seq[1:5])
# [2, 3, 7, 5]
```



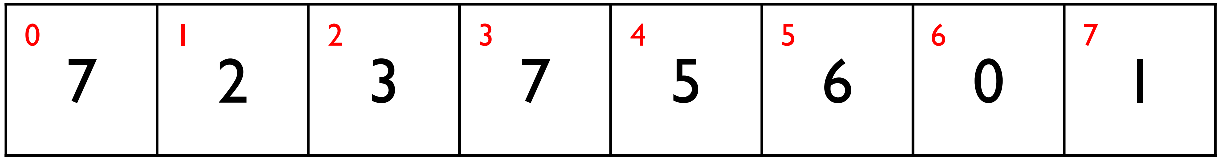
0	1	2	3	4	5	6	7
7	2	3	7	5	6	0	1

Slicing:

- Either start and stop can be omitted
  - start omitted: default to start from the beginning of the sequence
  - stop omitted: default to end of the sequence

```
print(seq[:5])
# [7, 2, 3, 7, 5]

print(seq[3:])
# [7, 5, 6, 0, 1]
```



0	1	2	3	4	5	6	7
7	2	3	7	5	6	0	1

Slicing:

- You can use negative indexes to start from the end of the sequence:
  - index -1 represents the last value of the sequence

```
print(seq[-4:])
# [5, 6, 0, 1]


print(seq[-6:-2])
# [3, 7, 5, 6]
```

0	1	2	3	4	5	6	7
7	2	3	7	5	6	0	1
-8	-7	-6	-5	-4	-3	-2	-1

Slicing:

- Return the last element of a sequence:

```java
// Java
int[] seq = {1, 2, 3, 4};
int last = seq[seq.length - 1];
```

```python
# Python
seq = [1, 2, 3, 4]
last = seq[-1]
```

- … Python = neat ☺

Final remarks:

- Remember that strings are sequences

```
s = "HELLO!"
print(s[2:4])
# "LL"
```

- Invalid indexes will generate an error

- Sets and dictionaries are not slice-able

Next lecture:

- Python functions