Recap and language details for Python exercises

2 - while and lists

While

while

```
while condition:
    instruction inside while
    ...
    instruction inside while
instructions after while
```

- The `condition` (Boolean expression) is evaluated.
- **Only if** the Boolean expression is `True` then the instructions within the `while` are executed (note the indentation).
- When you finish the instructions inside the while, **you go back to test the** `condition`
- If it is still `True`, you run the instructions inside the while again.
- It continues like this until the condition becomes `False`.
- If the condition is `False` you move on to the instructions after `while`.

while

```
while condition:
    inside while
```

- We can think of it as *repeat while the condition remains true*.

while

```
while condition:
    inside while
```

- We can think of it as *repeat while the condition remains true*.
- Attention: it is necessary to verify that the expression present in the condition is modified in the instructions inside the while.

`while`

```
while condition:
    inside while
```

- We can think of it as *repeat while the condition remains true*.
- Attention: it is necessary to verify that the expression present in the condition is modified in the instructions inside the while.
- Even though the shape is visually similar, the construct is very different semantically from the construct `if`.

while

```
while condition:
    inside while
```

- We can think of it as *repeat while the condition remains true*.
- Attention: it is necessary to verify that the expression present in the condition is modified in the instructions inside the while.
- Even though the shape is visually similar, the construct is very different semantically from the construct `if`.
- What happens when running this program?

```
a = 3
while a>0:
    print("Positive number")
```

Lists

List: definition and operators

Ordered and **mutable** sequences of elements.

- Elements within square brackets [1,2,3]
- Empty list: []
- List with only one item: [42]

Operations common to all sequences (already seen):

- Operator + to concatenate lists
- Operator * to repeat items (e.g. [1]*5 = ?)
- Operator [ ] to select individual elements (e.g. [1,2,3][0] = ?)
- Slicing Operator [:] and [::]
- Operator **in** and **not in**
- Function `len()`.
- Cycle **for** to iterate on the elements of the list

List: sequences mutable

Being mutable sequences they allow other operations.

- Assignment: `L[i] = 3`
- Assignment to a sublist: `L[1:3] = [28, 30]`
- Insertion of a sublist:

  ```
  >>> L = ["a", "d", "f"]
  >>> L[1:1] = ["b", "c"]
  >>> L
  ['a', 'b', 'c', 'd', 'f']
  ```

- Deletion of an item: **del** `L[i]` or of sublist: **del** `L[1:5]`
- Insert an element at the bottom of a list: `L.append(x)`
- ... but also `L += [x]`
- ... but also `L = L + [x]` (which however creates a copy...)
- Insert an item in a specific position `i`: `L.insert(s, x)`

List methods

What does this program print?

```
L = [1,2,3]
L = L.append(4)
print(L)
```

List methods

What does this program print?

```python
L = [1,2,3]
L = L.append(4)
print(L)
```

Many methods on lists, including `append`, `insert`, `extend`, `clear` modify the list (but do not change the identity) and return `None`.

What does `print(L.append(4))` print?

List: aliasing vs copies

- Objects and values:

```
>>> X = [1, 2, 3]
>>> Y = [1, 2, 3]
>>> X == Y
True
>> X is Y
False
```

- Create a copy of a list. Difference between:

```
A = [1, 2, 3]
B = A
```

e

```
A = [1, 2, 3]
B = A.copy() #equivalent: b = a[:]
```

- What is `B` referring to in the first case? And in the second?
  Click here to try on PythonTutor