More About Loops
### while loops

- The **while** loop is the simplest kind of loop
  - The *test* is performed before the statements are executed
    - Thus, if the *test* is initially **False**, the while loop does nothing
    - If the test is **True**, the statements are executed, then the *test* is performed again
  - The *statements* must eventually cause the test to become **False**, else you have a so-called infinite loop
A common idiom is to set something up before the while loop, and tweak it at the bottom of the loop.

- Get some value
  while something about the value:
    do some things with the value
    get another value

Example:
```python
password = input("Enter your password: ")
while password != actual_password:
    print("That’s not your password!")
    password = input("Enter your password: ")
```
for loops I

- **for** loops execute their statements for a fixed number of values, setting the *loop index* to each value in turn

- The values can be in the form of a *list*
  
  - `names = ["Tom", "Dick", "Harry"]`
    
    ```python
    for name in names:
        print(name)
    ```

- The values can be in the form of a *set*
  
  - `names = {"Tom", "Dick", "Harry"}`
    
    ```python
    for name in names:
        print(name)
    ```

- The values can be in the form of a *dictionary*
  
  - `names = {"Tom": 25, "Dick": 23, "Harry": 25}`
    
    ```python
    for name in names: # steps through the keys
        print(name, "->", names[name])
    ```
• **for** loops execute their statements for a fixed number of values, setting the **loop index** to each value in turn

• The values can be given by an **iterator**, which is a function that provides values as needed

• The most common iterator is **range**
  • **range**(start, end) produces integer values starting with **start** and going up to, but not including, **end**
  • **range**(end) is equivalent to **range**(0, end)
  • **range**(start, end, step) produces integer values starting with **start** and going up by steps of **step**, up to but not equalling or exceeding **end**

• **Example:**
  ```python
  for i in range(1, 10):
    print(i, i * i, i ** 3, i ** 4)
  ```
**break**

- The `break` statement is used to exit a loop early

- **Example:**
  ```python
  for i in range(1, 6):
      if i == 4:
          break
      print(i)
  ```

  produces  
  1
  2
  3

- If there is any reason to use a `break` that isn’t within an `if` statement, I can’t think of it

- Many programmers feel it is bad style to *ever* use a `break`

- I recommend using a `break` only as a last resort, if you can’t figure out a better way to exit a loop normally
**continue**

- The `continue` statement is used to skip the rest of the loop and go back to the top.

- **Example:**
  ```python
  for i in range(1, 6):
      if i == 4:
          continue
      print(i)
  ```

  produces:

  1
  2
  3
  5

- Like `break`, `continue` really only makes sense within an `if` statement.

- While not as bad as `break`, many programmers don’t like to use `continue`.

- Think about alternatives before using a `continue`
The `pass` statement is the easiest of all— it does nothing.

`pass` is used mostly as a placeholder, where a statement is required but you haven’t yet figured out what to do there.

Example:
```python
if illegal_alien(candidate):
    pass
else:
    hire(candidate)
```
Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retirer.

It seems that perfection is attained not when there is nothing more to add, but when there is nothing more to remove.

-- Antoine de Saint Exupéry