In this handout we look at some common use cases for **while** loops. These are problems where you *must* use a while loop because you don't know at the outset how many times you need to loop.

1 Polling pattern

The **polling pattern** arises when we want to ask the user for some input and keep re-prompting until we receive an input that is considered acceptable. For example, websites use this pattern when they want their users to create strong passwords that have a minimum length, at least one number, etc. The website prompts the user repeatedly until the user finally enters an acceptable password.

Consider this example: write a program that asks the user to type a four letter word – re-prompting them until they enter a string with four characters – and then prints a censored version of the word (asterisks for the middle two letters). There are three ways to solve this problem.

First approach: artificially cause while condition to be `True` so that loop is executed at least once:

```
word = '' # dummy value not 4 letters long
while len(word) != 4:
    word = raw_input("Enter word: ")
print "Censored word", word[0]+'**'+word[-1]
```

Second approach: copy some loop code and place it above the loop (a "loop and a half"):

```
word = raw_input("Enter word: ")
while len(word) != 4:
    word = raw_input("Enter word: ")
print "Censored word", word[0]+'**'+word[-1]
```

Third approach: use `True` as the while condition and rely on a **break** statement to exit the loop:

```
while True:
    word = raw_input("Enter word: ")
    if len(word) == 4:
        break
print "Censored word", word[0]+'**'+word[-1]
```

Use **break** sparingly as multiple breaks can make code difficult to read. You may not be allowed to use it on some exam problems or assignments.

In many settings, it's useful to create a separate function that checks the validity on the input and returns a boolean value indicating whether the input is acceptable. For example,

```
def is_valid(word):
    return len(word) == 4

word = raw_input("Enter word: ")
while not is_valid(word):
    word = raw_input("Enter word: ")
print "Censored word", word[0]+'**'+word[-1]
```

2 Sentinel pattern

In normal society, a sentinel is a guard; in the programming world, a **sentinel** is a special value that marks the end of an input sequence. The **sentinel pattern** arises when we want to ask the user for a *sequence* of inputs. For example, we might want to ask the user to enter a sequence of homework grades and then have the program compute the average grade. One way to solve this is have the user tell us, in advance, exactly how many grades they plan to enter. A more user-friendly way is to allow the user to type in a sentinel value to signal to the program that they are done. In this example, -1 might be a good sentinel since grades are usually non-negative!

Consider this example: write a program that asks the user to type in a sequence of numbers and then prints their sum. The user can enter 999 to signal the end of the input. The number 999 is the sentinel and should *not* be included in the sum. There are two ways to solve this problem.

First approach: use a "loop and a half":

```
total = 0
num = int(raw_input("Enter num (999 to quit): "))
while num != 999:
    total += num  # num must not be sentinel
    num = int(raw_input("Enter num (999 to quit): "))
print "Total is", total
```

Second approach: use `True` as the while condition and rely on a **`break`** statement to exit the loop:

```
total = 0
while True:
    num = int(raw_input("Enter num (999 to quit): "))
    if num == 999:  # check for sentinel
        break
    total += num
print "Total is", total
```

3 Exercises

Solutions are presented in class and also included in the moodle version of this handout.

1. Write a "guess a number" game. The computer chooses a secret number at random from 1 to 10. The user can make guesses and the computer gives hints ("higher" or "lower"). The game ends when the user guesses the number or gives up and types a negative number (a sentinel).

   Solution: First solution uses a loop and a half:

   ```
   import random
   secret = random.randint(1, 10)
   guess = int(raw_input("Guess: "))
   ```

```
while guess != secret and guess != -1:
    if guess > secret:
        print "Lower"
    else:
        print "Higher"
    guess = int(raw_input("Guess: "))

if guess == secret:
    print "You guessed it!"
else:
    print "You were so close!",
    print "The secret was", secret
```

Solution: An alternative solution uses a break statement:

```
import random
secret = random.randint(1, 10)

while True:
    guess = int(raw_input("Guess: "))
    if guess == -1 or guess == secret:
        break
    elif guess > secret:
        print "Lower"
    else:
        print "Higher"

if guess == secret:
    print "You guessed it!"
else:
    print "You were so close!",
    print "The secret was", secret
```

2. Take the sentinel example from Section 2 but calculate the *average* instead of the total.

Solution:

```
total = 0
count = 0
num = int(raw_input("Enter num (999 to quit): "))
while num != 999:
```

```
    # num must not be sentinel
    total += num
    count += 1
    num = int(raw_input("Enter num (999 to quit): "))
if count > 0:
    print "Average is", float(total)/count
else:
    print "You didn't enter any numbers, man!"
```

3. Previous problem but negative numbers are not allowed and should not be included in the average. Re-prompt the user if they are enter a negative number.

Solution:

```
total = 0
count = 0
num = int(raw_input("Enter num (999 to quit): "))
while num != 999:
    # num must not be sentinel
    if num >= 0:
        total += num
        count += 1
    else:
        print "Number must not be negative!"
    num = int(raw_input("Enter num (999 to quit): "))
if count > 0:
    print "Average is", float(total)/count
else:
    print "You didn't enter any numbers, man!"
```

Adapted from materials by Reges and Zingaro.