Loops

Loops are a way to repeatedly execute some code. Here's an example:

In [1]:
```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus
for planet in planets:
    print(planet, end=' ') # print all on same line
```

```
Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune
```

The `for` loop specifies

- the variable name to use (in this case, `planet`)
- the set of values to loop over (in this case, `planets`)

You use the word "`in`" to link them together.

The object to the right of the "`in`" can be any object that supports iteration. Basically, if it can be thought of as a group of things, you can probably loop over it. In addition to lists, we can iterate over the elements of a tuple:

In [2]:
```
multiplicands = (2, 2, 2, 3, 3, 5)
product = 1
for mult in multiplicands:
    product = product * mult
product
```

Out[2]:
```
360
```

You can even loop through each character in a string:

In [3]:
```
s = 'steganograpHy is the practicE of conceaLing a file, message, image, or
msg = ''
# print all the uppercase letters in s, one at a time
for char in s:
    if char.isupper():
        print(char, end='')
```

```
HELLO
```

range()

`range()` is a function that returns a sequence of numbers. It turns out to be very useful for writing loops.

For example, if we want to repeat some action 5 times:

```
ange(5):
```

```
    print("Doing important work. i =", i)
```

```
Doing important work. i = 0
Doing important work. i = 1
Doing important work. i = 2
Doing important work. i = 3
Doing important work. i = 4
```

`while` loops

The other type of loop in Python is a `while` loop, which iterates until some condition is met:

In [5]:
```
i = 0
while i < 10:
    print(i, end=' ')
    i += 1 # increase the value of i by 1
```

```
0 1 2 3 4 5 6 7 8 9
```

The argument of the `while` loop is evaluated as a boolean statement, and the loop is executed until the statement evaluates to False.

List comprehensions

List comprehensions are one of Python's most beloved and unique features. The easiest way to understand them is probably to just look at a few examples:

In [6]:
```
squares = [n**2 for n in range(10)]
squares
```

Out[6]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Here's how we would do the same thing without a list comprehension:

In [7]:
```
squares = []
for n in range(10):
    squares.append(n**2)
squares
```

Out[7]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

We can also add an `if` condition:

In [8]:
```
short_planets = [planet for planet in planets if len(planet) < 6]
short_planets
```

Out[8]:
```
['Venus', 'Earth', 'Mars']
```

(If you're familiar with SQL, you might think of this as being like a "WHERE" clause)

Here's an example of filtering with an `if` condition *and* applying some transformation to the loop variable:

In [9]:
```
# str.upper() returns an all-caps version of a string
loud_short_planets = [planet.upper() + '!' for planet in planets if len(plar
loud_short_planets
```

Out[9]:
```
['VENUS!', 'EARTH!', 'MARS!']
```

People usually write these on a single line, but you might find the structure clearer when it's split up over 3 lines:

In [10]:
```
[
    planet.upper() + '!'
    for planet in planets
    if len(planet) < 6
]
```

Out[10]:
```
['VENUS!', 'EARTH!', 'MARS!']
```

(Continuing the SQL analogy, you could think of these three lines as SELECT, FROM, and WHERE)

The expression on the left doesn't technically have to involve the loop variable (though it'd be pretty unusual for it not to). What do you think the expression below will evaluate to? Press the 'output' button to check.

In [11]:
```
[32 for planet in planets]
```

Out[11]:
```
[32, 32, 32, 32, 32, 32, 32, 32]
```

List comprehensions combined with functions like `min`, `max`, and `sum` can lead to impressive one-line solutions for problems that would otherwise require several lines of code.

For example, compare the following two cells of code that do the same thing.

In [12]:
```
def count_negatives(nums):
    """Return the number of negative numbers in the given list.

    >>> count_negatives([5, -1, -2, 0, 3])
    2
    """
    n_negative = 0
    for num in nums:
        if num < 0:
            n_negative = n_negative + 1
    n_negative
```

Here's a solution using a list comprehension:

In [13]:
```
def count_negatives(nums):
    return len([num for num in nums if num < 0])
```

Much better, right?

Well if all we care about is minimizing the length of our code, this third solution is better still!

In [14]:
```
def count_negatives(nums):
    # Reminder: in the "booleans and conditionals" exercises, we learned abo
    # Python where it calculates something like True + True + False + True t
    return sum([num < 0 for num in nums])
```

Which of these solutions is the "best" is entirely subjective. Solving a problem with less code is always nice, but it's worth keeping in mind the following lines from The Zen of Python:

> Readability counts.
> Explicit is better than implicit.

So, use these tools to make compact readable programs. But when you have to choose, favor code that is easy for others to understand.

Your Turn

You know what's next -- we have some **fun coding challenges** for you! This next set of coding problems is shorter, so try it now.
