Loops

Outline

Aesthetic: Eliminating Redundancy

We have seen code like this before:

```
player1 = get_player()
greet_player(player1)
player2 = get_player()
greet_player(player1)
player3 = get_player()
greet_player(player3)
```

What is the problem with this code?

Unbounded Iteration

What if we don't know the number of players in advance?

```
num_players = int(input("Players? "))
if num_players > 0:
    player1 = get_player()
if num_players > 1:
    player2 = get_player()
if num_players > 2:
    player2 = get_player()
# ... awkward, tricky, bad
```

Unbounded Programs

Think back to our calculator from last week—isn't it annoying to start it over from the console every time?

Some programs should never end, or shouldn't end until the user says so!

The `while` loop

Enter the `while` loop:

```
input_str = input("Next move? ")
while input_str != "quit":
    # ... play a round of the game ...
    input_str = input("Next move? ")
```

while again:

```
times = 0
while num > 0:
    num = (num + 1) // 2
    times += 1
print(f"{times} steps until zero")
```

Looping-until

- A `while` loop is made of two parts:
  - The *condition* is checked first of all, like `if`
  - If the condition is true, the *body* is executed
    - Otherwise we skip past the end of the loop
- After the body executes, control returns to the condition

Exercise

- How many times will the body of the loop execute?
- What will the value of x be at the end of each iteration of this loop?
- What will the value of x be when the loop has finished?

```
x = 0
while x + 17 < 30:
    x += 4
```

Exercise

This code is out of order and not indented properly. Reorder it to make it correctly compute the product of the first ten numbers starting from 1.

```
num += 1
num = 1
while num < 10:
product *= num
product = num
```

Exercise

Write a `while` loop that finds the first common multiple of two numbers `a` and `b`.

Hint: Define two variables `a_mult` and `b_mult`, initialized to `a` and `b`. `a_mult` will always be a multiple of `a`; likewise for `b_mult`. What does it mean if those two become equal?

The `for` loop

Python has a second type of loop: the `for` loop.

```
for i in range(5):
    print(i)
```

Looping-over

- `for` loops always try iterate a specific number of times.
- We are looping *over*, rather than looping *until*.
- Instead of a condition, for loops:
  - Define a *loop variable* (`i`)...
  - a *loop iterator* (`range(5)`)...
  - and a body

Exercise

- How many times will the body of the loop execute?
- What will the value of x be at the end of each iteration of this loop?
- What will the value of x be when the loop has finished?

```
x = 0
for y in range(5):
    x += y
```

range

`range` is the only kind of iterator we'll discuss for now.

`range` can get pretty fancy, specifying start, end, and step-size:

```
for i in range(5): print(i)
for i in range(1, 5): print(i)
for i in range(0, 7, 2): print(i)
for i in range(7, 0, 2): print(i)
for i in range(7, 0, -2): print(i)
```

Early exit

- Loops like `for` or `while` usually end naturally
  - When the iterator is empty
  - When the condition becomes false
- But if we `return` from the function or `break` out of the loop...
  - The loop will end immediately

Exercise

- How many times will `step` be printed?
- What value will be printed for `x` at the end?

```
x = 10
for i in range(5):
    x += i
    if x > 15:
        break
    print("step")
print(x)
```

Exercise

- How many times will `step` be printed?
- What value will be printed for `x` at the end?
- What does this tell us about how `range` works?

```
x = 5
for i in range(x):
  x += i
  print("step")

print(x)
```

Quiz

Truncating vs Float Division

- Remember the difference between `/` and `//`:
  - `5 / 2` is 2.5, but `5 // 2` is 2
- This will be useful for the assignment

Commenting Complex Code

When we have multiple-step functions, it's a good idea to intersperse comments:

```
x = y ** 2
z = 2 + x / 5
# Compute manifold variance
for i in range(x):
    # first, frobulate the chunks:
    z += f(i) + f(x)
    x = z * 2
    # then invert the remainder
    z = -(1 / z)
```

It's up to you when to write comments versus defining helper functions.

Testing Your Code

How do you know your code works?

One way to build confidence is to test with specific inputs.

You can do this in various ways:

```
def my_fun(x):
    # ...


print(my_fun(5)) # should be 17
print(my_fun(12)) # should be 9
print(my_fun(20)) # should be 142
```

Testing Your Code

It may be better to split the tests from the code:

- In `code.py`:

```
def my_fun(x):
    # ...
```

- In `test.py`:

```
import code
if code.my_fun(5) != 17:
    print("wrong for 5")
if code.my_fun(12) != 9:
    print("wrong for 12")
# etc
```

Then we can run `python3 test.py` to get some information.

Testing Your Code

The starter code takes this one step further by using the built-in `assert` statement, which will terminate the program if its input is `False`.

Testing, of course, can only reveal the presence of bugs and never their absence (thanks to E.W. Dijkstra, whose name you'll learn in CSCI 062 or 140, for this insight).

In fact, writing tests before writing your code may be a useful way to know what the code will need to do, and to try to build up and understand the relation between inputs and outputs.