Strings, Lists, and Iteration

Camp Inc. Coding Track

Summer 2016

Review: Branching

```
if condition:
    # do something
elif other_condition:
    # do something else
else:
    # do something else
```

Review: Branching

```
if condition:
    # do something
elif other_condition:
    # do something else
else:
    # do something else
```

Example Program

```
name = input('What is your name? ')
if name == 'Jack':
    print('Your name is the best!')
elif len(name) == 4:
    print('Your name is 4 letters!')
else:
    print('Pleased to meet you', name)
```

Short-Circuiting Operators

What if you want to test the existence of multiple conditions? This is what `and` and `or` are for.

```
if i == 4 or j == 7:
    i = j - 1
    j = i + 1
if i == 1 and j == 1:
    print('Hello World!')
```

Short-Circuiting Operators

What if you want to test the existence of multiple conditions? This is what `and` and `or` are for.

```
if i == 4 or j == 7:
    i = j - 1
    j = i + 1
if i == 1 and j == 1:
    print('Hello World!')
```

`and` and `or` are capable of **short-circuiting** your program. This means not all conditions will be evaluated if they don't need to.

Review: More Comparison Operators

`<`	Less than	`<=`	Less or equal	`!=`	Not equal
`>`	Greater than	`>=`	Greater or equal	`is`	Same instance

Review: More Comparison Operators

`<`	Less than	`<=`	Less or equal	`!=`	Not equal
`>`	Greater than	`>=`	Greater or equal	`is`	Same instance

Example Program

```
age = int(input('What is your age? '))
if age < 0:
    print('I don\'t think so')
elif age <= 10:
    print('Wow! You\'re young!')
elif age != 16:
    print('Cool cool.')
else:
    print('Sweet sixteen.')
```

Looping on a Condition

`while` is a loop which checks a condition before entry into a code section, and repeats **while** the condition is `True`.

```
i = 1
while i <= 10:
    print(i, 'Hello, World!')
    i = i + 1
```

Trace Me Some Loops

```
x = 1
i = 2
while x < 10:
    x = x + i
    i = i * 2
    print(x, i)
```

Trace Me Some Loops

```
x = 1
i = 2
while x < 10:
    x = x + i
    i = i * 2
    print(x, i)
```

```
3 4
7 8
15 16
```

Trace Me Some Loops

```
i = 0
while i < 2:
    i = i + 1
    j = 0
    while j < 3:
        j = j + 1
        print(i, j)
```

Trace Me Some Loops

```
i = 0
while i < 2:
    i = i + 1
    j = 0
    while j < 3:
        j = j + 1
        print(i, j)
```

```
1 1
1 2
1 3
2 1
2 2
2 3
```

Trace Me Some Loops

```
i = 0
while i < 20:
    if i % 3 == 0:
        print(i)
    i = i + 1
```

Trace Me Some Loops

```
i = 0
while i < 20:
    if i % 3 == 0:
        print(i)
    i = i + 1
```

```
0
3
6
9
12
15
18
```

Lists

Lists are a data structure which allow us to store **ordered data**. We can specify list literals in Python using brackets.

```
my_list = [1, 2, "Hello", "CI"]
```

Lists

Lists are a data structure which allow us to store **ordered data**. We can specify list literals in Python using brackets.

```
my_list = [1, 2, "Hello", "CI"]
```

Specific members of a list can be accessed by specifying the **zero-indexed** offset in brackets.

- `my_list[2]` is **?**

Lists

Lists are a data structure which allow us to store **ordered data**. We can specify list literals in Python using brackets.

```
my_list = [1, 2, "Hello", "CI"]
```

Specific members of a list can be accessed by specifying the **zero-indexed** offset in brackets.

- `my_list[2]` is `"Hello"`

Lists

Lists are a data structure which allow us to store **ordered data**. We can specify list literals in Python using brackets.

```
my_list = [1, 2, "Hello", "CI"]
```

Specific members of a list can be accessed by specifying the **zero-indexed** offset in brackets.

- `my_list[2]` is `"Hello"`
- `my_list[0]` is **?**

Lists

Lists are a data structure which allow us to store **ordered data**. We can specify list literals in Python using brackets.

```
my_list = [1, 2, "Hello", "CI"]
```

Specific members of a list can be accessed by specifying the **zero-indexed** offset in brackets.

- `my_list[2]` is `"Hello"`
- `my_list[0]` is `1`

Why Zero-Indexed?

In lower-level programming languages, lists are stored as simply a base address in memory, and the value in the brackets is the offset. The offset is added to the base address to find the memory address of the item.

Address	$\cdots$	1350	1351	1352	1353	1354	$\cdots$
Value	$\cdots$	12	13	7	21	3	$\cdots$

Iterating Over a List using `while`

Using what we know about **`while`** loops, we can iterate over a list using a counter variable. Here is an example:

```
i = 0
while i < len(my_list):
    print(my_list[i])
    i = i + 1
```

Iterating Over a List using `while`

Using what we know about **`while`** loops, we can iterate over a list using a counter variable. Here is an example:

```
i = 0
while i < len(my_list):
    print(my_list[i])
    i = i + 1
```

If `my_list` was `[1, 2, "Hello", "CI"]`, then this would print:

```
1
2
Hello
CI
```

Iterating Over a List using `for`

Python provides a clean **range-based** construct for iterating over **iterables** called `for`. Here's it's syntax:

```
for var_name in iterable:
    # do something
```

Iterating Over a List using `for`

Python provides a clean **range-based** construct for iterating over **iterables** called `for`. Here's it's syntax:

```
for var_name in iterable:
    # do something
```

So here is an example of iterating over our previous list:

```
for item in my_list:
    print(item)
```

```
1
2
Hello
CI
```

Generating Ranges

The generator function `range` creates an iterable for looping over a sequence of numbers. This may be more convenient than using `while`.

The syntax is `range(`$start$`, `$stop$`, `$step$`)` if $start$ is not provided, it is assumed to be 0, and if $step$ is not provided, it is assumed to be 1.

Here is an example:

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

```
0
1
2
3
4
```

Strings are Iterables!

```
for c in 'hello world':
    print(c)
```

```
h
e
l
l
o

w
o
r
l
d
```

.split()ting Strings

To separate a string based on white spaces, call `.split()` on it. Here is an example:

```
my_str = '  camp inc is  really cool'
wordlist = my_str.split()
# wordlist should be ['camp', 'inc', 'is', ... ]
for word in wordlist:
    print(word)
```

```
camp
inc
is
really
cool
```

.split()ting Strings

To separate a string based on white spaces, call `.split()` on it. Here is an example:

```
my_str = '  camp inc is  really cool'
wordlist = my_str.split()
# wordlist should be ['camp', 'inc', 'is', ... ]
for word in wordlist:
    print(word)
```

The . Operator

The . operator used above is actually the **accessor operator**, however, most programmers simply call it the **dot operator**. It allows us to use a function which is specific to a certain data type on the object.

Concatenating Strings

To chain strings together, end to end, use the + operator.

```
H = input('what is the hour? ')
M = input('what is the minute? ')
print('The time is now ' + H + ':' + M + '.')
```

```
what is the hour? 13
what is the minute? 40
The time is now 13:40.
```

.format()ting Strings

It may be more convenient to specify the format for a string, then list the items to put in after.

```
H = input('what is the hour? ')
M = input('what is the minute? ')
print('The time is now {}:{}.'.format(H,M))
```

```
what is the hour? 13
what is the minute? 40
The time is now 13:40.
```