Week 2: Branching & Iteration

2023-09-04

Agenda

- Strings
- Comparison operators
- Iteration and loops: `for` and `while`
- Indentation
- Control flow: `if`, `elif`, `else`

string data type

- Text, letter, character, space, digits (numeric characters), etc.
- Create with single or double quotes (but every instance needs to be consistent in itself)

```
greeting = "Hello! How are you"
who = 'Anastasia'
key_str = '1111'
key_integer = 1111
```

string data type

- Concatenate with +

```
print(greeting + ' ' + who + '?')
print('The passcode is ' + key_str)
print('The passcode is ' + str(key_integer))
```

```
Hello! How are you Anastasia?
The passcode is 1111
The passcode is 1111
```

string data type

- Even, do this:

```
who * 3
```

```
'AnastasiaAnastasiaAnastasia'
```

`string` data type

- Strings can also be created with `'''` or `"""`
- These can handle multi-line strings (but don't have to be multiline)

```
my_string = '''
This is a string. It is spanning
multiple lines.
'''
print(my_string)
```

```
This is a string. It is spanning
multiple lines.
```

string data type

```python
my_other_string = """I could do this too."""
print(my_other_string)
```

```
I could do this too.
```

`string` data type

- Tip: to create a string with double quotes in it, create it with single quotes (and vice versa)

```python
convo_1 = '"Which is worse, ignorance or apathy?"'
print(convo_1)
convo_2 = "'Which is worse, ignorance or apathy?'"
print(convo_2)
```

```
"Which is worse, ignorance or apathy?"
'Which is worse, ignorance or apathy?'
```

string data type

- If you need both, `'''` can be used

```
convo_3 = '''"What's the matter?"'''
print(convo_3)
```

```
"What's the matter?"
```

print() : Print to output cells (or consoles)

- print() can print strings and other things together

```
n_apples = 3 # n_apples is an integer
print('I ate', n_apples, 'apples.')
```

```
I ate 3 apples.
```

- Or we could combine them to a string first (but n_apples will have to be converted to a string)

```
print('I ate ' + str(n_apples) + ' apples.')
```

```
I ate 3 apples.
```

Comparison Operators

- Used to compare to variables to one another
- These evaluate to a Boolean:

```
var1 > var2
var1 >= var2
var1 < var2
var1 <= var2
var1 == var2
var1 != var2
```

Comparison Operators

- This is widely useful, including
  - Control flow (we will see this in a bit)
  - Filter data
  - And many many more

Logical Operators on Booleans

- **`not`**, **`or`**, **`and`** are special words for logical operators
- **`not a`** -> True if **a** is False; False if **a** is True
- **`a or b`** -> True if at least one of **a** or **b** is True
- **`a and b`** -> True if both are True

Examples

```python
hours = 20
print(hours > 24) # More than a day?
```

```
False
```

Examples

```
bike = True # How did you commute today?
bus = False

print(not bike)
print(not bus)
print(bike or bus)
```

```
False
True
True
```

```
print(bike and bus)
```

```
False
```

Control Flow: Branching

- **`if`** is used to evaluate an expression **if** a condition is **True**

```
if <condition>:
    <expression>
    <expression>
```

Note on Indentation

- Be careful and meticulous when you indent multiple times (and multiple levels)

```
if <condition>:
    <expression>
    <expression>
    <expression>

<expressions that don't depend on condition>
```

- The expressions should (by convention) be indented by 4 spaces or a Tab
- That's how Python understands that those are the expressions to be run if the condition is True

Note on Indentation

Best practices to avoid headaches:

- Either always use Tabs or always spaces in a Python script
- Python community mostly uses 4 spaces
- Most IDEs will allow you to automatically convert Tabs to 4 spaces
- Applicable not only to branching but also to: `for`, `while`, `with`, `def`, `class`, `try`, `except`, etc.

For more discussion see here.

Control Flow: Branching

- We can also use `else` with `if`.
- Evaluate `expression1` if condition is True, otherwise evaluate `expression2`.

```
if <condition>:
    <expression1>
else:
    <expression2>
```

Example

```python
number = 12

if number % 2 == 0:
    print("Number is even.")
else:
    print("Number is odd.")
```

```
Number is even.
```

Control Flow: Branching

- `elif` stands for else if
- If condition1 is True, evaluate expression1
- If condition1 is not True but condition2 is True, evaluate expression2

```
if <condition1>:
    <expression1>
elif <condition2>:
    <expression2>
elif <condition3>:
    <expression3>
    .
    .
    .
else:
    <last-expression>
```

Control Flow: Branching

- Note that, in this setting, an expression is only evaluated only if everything above is False
- E.g. expression3 will not be evaluated if expression1 and expression2 are not **both** False

```
if <condition1>:
    <expression1>
elif <condition2>:
    <expression2>
elif <condition3>:
    <expression3>
    .
    .
    .
else:
    <last-expression>
```

Example

```
number = 7

if number > 0:
    print("Positive number")
elif number == 0:
    print('Zero')
else:
    print('Negative number')
```

```
Positive number
```

Further Note on Indentation

- Many nested indentations are normal and common

```
number = 12
if number % 2 == 0:
    print("Number is even.")
    if number % 3 == 0:
        print("Number is divisible by 3.")
    else:
        print("Number is not divisible by 3.")
else:
    print("Number is odd.")
```

```
Number is even.
Number is divisible by 3.
```

- Note how indentation determines which `else` belongs to which `if`

Control Flow: while Loops

- Evaluate the expressions as long as condition is True

```
while <condition>:
    <expression>
    <expression>
    ...
```

Example

```
# program to display numbers from 1 to 5

# initialize the variable
i = 1
n = 5

# while loop from i = 1 to 5
while i <= n:
    print(i)
    i = i + 1
```

```
1
2
3
4
5
```

Another Example

```
# smallest number greater than 700 divisible by 13
number = 700
while not number % 13 == 0:
    print(number, "is not divisible by 13.")
    number = number + 1
print(number, "is divisible by 13.") # 702/13 = 15
```

```
700 is not divisible by 13.
701 is not divisible by 13.
702 is divisible by 13.
```

Example

- `x = x + 1` can be shortened to `x += 1`

```python
number = 700
while not number % 13 == 0:
    number += 1
print(number)
```

```
702
```

Control Flow: for Loops

- Useful when the number of iterations is known
- Its function can be achieved by a `while` loop, but `for` loop is easier
- Every time through the loop, `<variable>` assumes a new value (iterating through `<iterable>`)

```
for <variable> in <iterable>:
    <expression>
    <expression>
    ...
```

Control Flow: for Loops

- Iterable is usually `range(<some_num>)`
- Can also be a list, tuple, string, dictionary etc.

```
for <variable> in range(<some_num>):
    <expression>
    <expression>
```

range()

`range(start, stop, step)`

- `start = 0` and `step = 1`
- Only `stop` is required
- It will start at 0, loop until `stop - 1`

Example

- If we supply one argument, it's understood as the `stop` argument

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

```
0
1
2
3
4
```

Example

```
for i in range(11, 15):
  print(i)
```

```
11
12
13
14
```

Example

```python
for i in range(10, 30, 5):
  print(i)
```

```
10
15
20
25
```

Example

```
for i in range(10, 30, 5):
  print(i % 10)
```

```
0
5
0
5
```

Can iterate over other things too

```
for char in 'CT DS DHCSS':
    print(char + "!")
```

```
C!
T!
 !
D!
S!
 !
D!
H!
C!
S!
S!
```

Break statement

- Exits the loop it is in
- Remaining expressions are not evaluated
- In nested loops, only innermost loop exited

```
for i in range(1, 4):
    for j in range(1, 3):
        if i == 2 and j == 2:
            break
        print(i, j)
```

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

Continue statement

- Continues to the next iteration of the loop, but does not exit loop
- Remaining expressions are not evaluated

```python
for i in range(1, 4):
    for j in range(1, 3):
        if i == 2 and j == 2:
            continue
        print(i, j)
```

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

Break and Continue

- `break` and `continue` can be used in both for and while loops

```python
var = 5
while var > 0:
    var -= 1
    print(var)
    if var == 3:
        continue
    if var == 2:
        break
    print('Current variable value :', var)
print("Good bye!")
```

```
4
Current variable value : 4
3
2
Good bye!
```