Intro to Python (Class 4)

Ben Bettisworth

Section 1

Loops

Loops

Naturally, when you have a list, you will want to do *something* with every item in the list. For this, we have loops.

Loops

There are 2 keywords associated with looping in python:

- `while`
- `for`

While loops

While loops repeat the body of the loop *while* some condition is true.

Example

```
a = 20
b = 3
while a < b:
  a = a / 2
```

While loops

In general, `while` are used when you don't know how long the loop will last. For most other loops, you know how long loop will last.

Example

```
my_list = [1,2,3]
index = 0
result = 0
while index < len(my_list):
  result += my_list[index] + my_list[index] ** 0.5
  index += 1
```

Determining the length in python

Python has the special function `len` to tell you the length of things (`list`, `tuple`, etc).

Example

```
l = [0,1,2]
len(l)
```

Output

```
3
```

Determining the length in python

The function `len` gives *the number of items* in the list. This means that the last valid index is `len(list) - 1`.

Example

```
my_list = [1,2,3]
index = 0
result = 0
while index < len(my_list):
  result += my_list[index] + my_list[index] ** 0.5
  index += 1
```

Exercise

Collatz proposed a iterative process. Given a number n, if n is even, then return $n/2$. Otherwise return $3n + 1$.

Collatz Conjecture

Collatz's conjecture is that this process will eventually reach 1, for any given positive n.

Implement a function using a while loop which counts the steps required to reach 1 for a given n.

Exercise

(A Possible) Solution

```
def collatz(n):
  steps = 0
  while n != 1:
    steps += 1
    if n % 2 == 0:
      n /= 2
    else:
      n = 3 * n + 1
  return steps
```

For loops

The pattern of iterating over a list of values is so common, that there is a special loop for it. The `for` loop executes the body of the loop once *for every* item in the list.

Example

```
my_list = [1,2,3]
result = 0
for item in my_list:
  result += item + item ** 0.5
```

Exercise

Write a function that takes a list of numbers and a threshold and counts the number of items in the list which is below that threshold.

Exercise

(A Possible) Solution

```
def count_char(numbers, threshold):
  counter = 0
  for item in numbers:
    if item < threshold:
      counter += 1
  return counter
```

Range loops

Iterating over consecutive integers is so common, there exists a special function to generate this list called `range`.

Example

```
result = 0
for i in range(1,4):
  result += i + i ** 0.5
```

Range loops

There are 3 ways to call range. This different ways of calling range are distinguished by the *number of arguments*.

- `range(stop)` = `[0, ..., stop - 1]`
- `range(start, stop)` = `[start, ..., stop - 1]`
- `range(start, stop, step)` = `[start, start + step, ..., stop - 1]`

Range Loops

`range(stop) = [0, ..., stop - 1]`

One Argument

```
for i in range(3):
  print(i, "foo" * i)
```

Output

```
0
1 "foo"
2 "foofoo"
```

Range Loops

Two Arguments

```
for i in range(1,3):
  print(i, "foo"*i)
```

Output

```
1 "foo"
2 "foofoo"
```

Range Loops

Three Arguments

```
for i in range(1,5,2):
  print(i, "foo"*i)
```

Output

```
1 foo
3 foofoofoo
```

Exercise

Write a function which takes an integer n and prints all the even numbers which are less than n.

Extension: Write a function that takes an integer n and returns all the *prime* numbers which are less than n.

Exercise

Code

```
def find_evens(n):
  for i in range(n):
    if i % 2 == 0:
      print(i)

find_evens(6)
```

Output

```
0
2
4
```

Exercise

```
def find_primes(n):
  primes = []
  for i in range(2,n):
    is_prime = True
    for p in primes:
      if i % p == 0:
        is_prime = False
        break
    if is_prime:
      primes.append(i)
  return primes
```

Nesting loops

Loops can be nested, that is they can be arranged like this

Code

```
for i in range(2):
  for j in range(2):
    print(f"i: {i}, j:{j}")
```

Output

```
i: 0, j:0
i: 0, j:1
i: 1, j:0
i: 1, j:1
```

Exercise

Write a function that draws a rectangle that is n by m characters in size. To print a single character to the screen, use `print("-", end="")`.

Exercise

(A Possible) Solution

```
def draw_rect(n,m):
  for i in range(n):
    for j in range(m):
      if i == 0 or i == n-1:
        print("-", end="")
      elif j == 0 or j == m-1:
        print("|", end="")
      else:
        print(" ", end="")
    print("")
```

Section 2

Slicing

Slicing

List in python can be *sliced* into smaller lists. The syntax for this is `list[start:stop]`. As always, the stop index is one *after* the last index.

Example

```
values = [0,1,2,3,4]
print(values[1:3])
```

Output

```
[1,2]
```

Slicing

Slice indices do not need to be less than the list length.

Example

```
values = [0,1,2,3,4]
print(values[5:10])
```

Output

```
[]
```

Unbounded Slicing

One of the endpoints can be left out from the slice indices. Given a list `values = [1,2,3,4,5]`.

No begin

```
values[:3]
```

Output

```
[1, 2, 3]
```

No end

```
values[3:]
```

Output

```
[4, 5]
```

Exercise

Write a function that takes a list and returns two slices of equal(ish) length.

Extension: Write a function that takes a list `values`, and returns a list of all possible slices of `values`.

Exercise

Code

```
def half_list(values):
  midpoint = len(values) // 2
  return (values[:midpoint], values[midpoint:])

half_list([1,2,3,4,5,6])
```

Output

```
([1, 2, 3], [4, 5, 6])
```

Section 3

Dictionaries

Maps

Maps are a conceptual object which operate on a *key-value* system. You give a map a *key*, and it gives back a *value*.

In Python, maps are generally realized as dictionaries.

Properties of Keys

There are a few properties of keys that are good to know.

- Keys must be *unique* within a map.
- Each key indicates a single *value*.
- In Python, keys must be *hashable*.

What does “Hashable” Mean?

In computer science, a *hash function* is a function which takes some value, and turns it into a number. This number is then used to look-up the location of the value.

Hashable in Python

In Python, a *type* is *hashable* if:

- It is a primitive (int, float, str, etc)
- It implements the methods
  - `__hash__()`
  - `__eq__()`
- It is a tuple of hashable types.

Creating a dictionary

Dictionaries can be created with:

- The `dict()` function, and
- the `{}` syntax.

Example

```
dict1 = dict()
dict2 = {}
```

Adding to a Dictionary

```
dict[key] = value
```

Example

```
d = {}
d["hello"] = "world"
print(d)
```

Output

```
{'hello': 'world'}
```

Getting values from a Dictionary

Values can be accessed in two ways:

- `dict[key]` (Will throw an error if there is no key)
- `dict.get(key)` (Will return `None` if there is no key)

Example

```
d = {}
d["hello"] = "world"
d["foo"] = "bar"
print(d["foo"])
```

Output

```
bar
```

Removing items from a Dictionary

Use `dict.pop(key)`

Example

```
d = {}
d["hello"] = "world"
d["foo"] = "bar"
d.pop("hello")
print(d)
```

Output

```
{'foo': 'bar'}
```

Loops and Dictionaries

There are three methods that are of interest:

- `dict.keys()` which returns a list of keys.
- `dict.values()` which returns a list of values.
- `dict.items()` which returns a list of `(key,value)`.

The lists returned by these functions will be in insertion order

Looping Examples

For the following examples, suppose we have the following dictionary.

```
example_dict = {}
example_dict["foo"] = "bar"
example_dict["buzz"] = "bar"
example_dict["hello"] = "world"
example_dict["π"] = 3.14
example_dict[1.618] = "ϕ"
example_dict["my name is"] = "ben"
```

Looping over Keys

Code

```
for key in example_dict.keys():
  print(f"the current key is '{key}'")
```

Output

```
the current key is 'foo'
the current key is 'buzz'
the current key is 'hello'
the current key is 'π'
the current key is '1.618'
the current key is 'my name is'
```

Looping over Values

Code

```
for value in example_dict.values():
  print(f"the current value is '{value}'")
```

Output

```
the current value is 'bar'
the current value is 'bar'
the current value is 'world'
the current value is '3.14'
the current value is 'ϕ'
the current value is 'ben'
```

Looping over Items

Code

```
for key, value in example_dict.items():
  print(f"key: '{key}', value: '{value}'")
```

Output

```
key: 'foo', value: 'bar'
key: 'buzz', value: 'bar'
key: 'hello', value: 'world'
key: 'π', value: '3.14'
key: '1.618', value: 'ϕ'
key: 'my name is', value: 'ben'
```

Exercise

Write a function that take a dictionary, and outputs the key corresponding to the largest string in the dictionary. You can check that a variable is a string with

```
type(var) is str
```

Extension: Return a list of keys sorted by length of the value.

Exercise

(A Possible) Solution

```
def longest_string(input_dict):
  max_length = 0
  max_key = None
  for key, value in input_dict.items():
    if type(value) is str:
      if max_length < len(value):
        max_key = key
        max_length = len(value)

  return max_key
```

Output

```
>>> longest_string(example_dict)
'hello'
```

Sets

Sets are a collection of *unique* values. In Python, sets are also dictionaries without values.

Example

```
s = set()
s.add("foo")
s.add("bar")
print(s)
```

Output

```
{'foo', 'bar'}
```

Uniqueness of Values in Sets

The defining property of sets is they only contain a value *once*.

Code

```
s = set()
a = "foo"
b = a
s.add(a)
s.add(b)
print(s)
```

Output

```
{'foo'}
```

Operations on Sets

Sets support the important (math) set operations:

- Union (`a.union(b)` / `a | b`)
- Intersection (`a.intersection(b)` / `a & b`)
- Difference (`a.difference(b)` / `a - b`)
- Symmetric Difference (`a.symmetric_difference(b)` / `a ^ b`)