Using Iterables

Collections in Python

Digital Career Institute

Using Iterables

Iterables, as the name suggests, are used for one main purpose: **iteration**.

There are also a variety of built-in functions that require iterables as arguments.

A list of strings can be iterated to access each one of its values and the `len` function can be used to obtain the number of items in the passed iterable (in this case, a string).

The same applies to every type of iterable.

```
>>> iterables_list = [
...     "string", "list", "set",
...     "tuple", "dictionary"
... ]
>>> for item in iterables_list:
...     print(item, len(item))
...
string 6
list 4
set 3
tuple 5
dictionary 10
```

Iterating Dictionaries

Digital Career Institute

Dictionaries are a special case, because each element is a composite of a key and a value.

The dictionary iterator yields only the key.

It has the same effect as using the `keys` method of the dictionary. This method yields the key of each item in the dictionary.

Dictionaries also have other methods to yield the values.

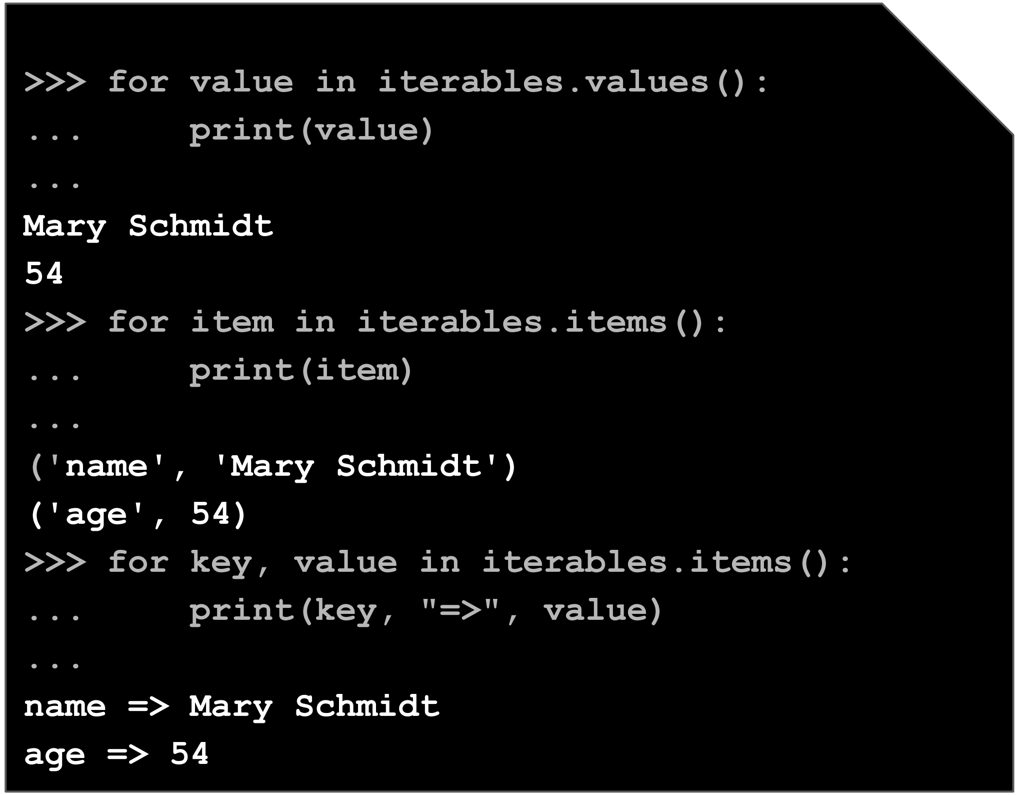
```
>>> profile = {
...     "name": "Mary Schmidt",
...     "age": 54
... }
>>> for key in iterables:
...     print(key)
...
name
age
>>> for key in iterables.keys():
...     print(key)
...
name
age
```

Iterating Dictionaries

The method **`values`** will yield the values of each item.

The method **`items`** will yield a tuple containing the key and value of each item.

The tuple can be unpacked in the same **`for`** instruction to make the code more readable.



```
>>> for value in iterables.values():
...     print(value)
...
Mary Schmidt
54
>>> for item in iterables.items():
...     print(item)
...
('name', 'Mary Schmidt')
('age', 54)
>>> for key, value in iterables.items():
...     print(key, "=>", value)
...
name => Mary Schmidt
age => 54
```

Iterating Tuples

Another common pattern is to use tuples instead of dictionaries to store **key-value** pairs that need to remain constant. This can be done defining a bi-dimensional tuple (a tuple of tuples).

The iteration will yield a tuple that can be unpacked like we do with the `items` method of a dictionary.

```
>>> days = (
...         ('Mon', 'Monday'),
...         ('Tue', 'Tuesday'),
...         ('Wed', 'Wednesday'),
...         ('Thu', 'Thursday'),
...         ('Fri', 'Friday'),
...         ('Sat', 'Saturday'),
...         ('Sun', 'Sunday')
... )
>>> for key, value in days:
...         print(key, "=>", value)
...
Mon => Monday
Tue => Tuesday
# continues
```

Functions with Iterables: Enumerate

Python has some built-in functions that require or accept iterables and can be very useful in common situations. `list`, `tuple`, `dict` and `set` are some of them.

Another one of the most used functions is `enumerate`.

The `enumerate` function takes any iterable and for each value yields a tuple containing the position of that value and the value itself.

```
>>> list = [
...         'Monday',
...         'Tuesday',
...         'Wednesday',
...         'Thursday',
...         'Friday',
...         'Saturday',
...         'Sunday'
... ]
>>> for position, value in enumerate(list):
...         print(position, "=>", value)
...
0 => Monday
1 => Tuesday
# continues
```

Functions with Iterables: Enumerate

The **`enumerate`** function allows for a simpler and more readable code to the alternative of adding a counter manually.

```
>>> list = [
...       'Monday',
...       'Tuesday',
...       'Wednesday',
...       'Thursday',
...       'Friday',
...       'Saturday',
...       'Sunday'
... ]
>>> position = 0
>>> for value in list:
...       print(position, "=>", value)
...       position += 1
...
0 => Monday
1 => Tuesday
# continues
```

Functions with Iterables: Enumerate

The **`enumerate`** function can even be used on iterables that yield iterables, like the **`items`** method of a dictionary.

This iterable can be further unpacked using parentheses.

```
>>> dict = {
...       "name": "Mary Schmidt",
...       "age": 54
... }
>>> for position, value in enumerate(dict.items()):
...       print(position, "=>", value)
...
0 => ('name', 'Mary Schmidt')
1 => ('age', 54)
>>> for pos, (key, val) in enumerate(dict.items()):
...       print(pos, ".", key, "=>", val)
...
0 . name => Mary Schmidt
1 . age => 54
```

Functions with Iterables: Zip

The **`zip`** function takes any number of iterables and returns a **`zip object`**.

This object is an iterator that yields a tuple with the value in each position of each passed iterable.

```
>>> nums = [1, 2, 3]
>>> eng = ("one", "two", "three")
>>> zip1 = zip(nums, eng)
>>> print(zip1)
<zip object at 0x7ff4f285cd80>
>>> for item in zip(nums, eng):
...     print(item)
...
(1, 'one')
(2, 'two')
(3, 'three')
```

Functions with Iterables: Zip

Any kind of iterable (and any amount) can be passed to the function.

Notice that sets can be passed too, but since they have no order the items are fetched randomly.

Every time we execute this code the german words will appear in a different order.

```
>>> nums = "123"
>>> eng = ("one", "two", "three")
>>> deu = {"ein", "zwei", "drei"}
>>> cat = {"un": 1, "dos": 2, "tres": 3}
>>> fra = ["un", "deux", "trois"]

>>> for item in zip(nums, eng, deu, cat, fra):
...     print(item)
...
('1', 'one', 'zwei', 'un', 'un')
('2', 'two', 'drei', 'dos', 'deux')
('3', 'three', 'ein', 'tres', 'trois')
```

Functions with Iterables: Zip

Digital Career Institute

Items can also be unpacked to provide a more readable code inside the loop.

```
>>> nums = "123"
>>> eng = ("one", "two", "three")
>>> deu = {"ein", "zwei", "drei"}
>>> cat = {"un": 1, "dos": 2, "tres": 3}
>>> fra = ["un", "deux", "trois"]
>>> for num, en, de, ca, fr in zip(nums, eng, deu, cat, fra):
...             nums, eng, deu, cat, fra):
...     print(num + ". " + en, de, ca, fr)
...
1. one drei un un
2. two zwei dos deux
3. three ein tres trois
```

Functions with Iterables: Sum, Max & Min

Some of the built-in mathematical functions accept iterables as arguments, so long as the iterable yields numeric values.

`Max` and **`min`** also accept string iterables, as the `>` operand also exists for this type.

```
>>> nums = [1, 2, 3, 4, 5]
>>> print(sum(nums), max(nums), min(nums))
15 5 1
>>> nums = {1, 2, 3, 4, 5}
>>> print(sum(nums), max(nums), min(nums))
15 5 1
>>> nums = [1, 2, 3, 4, "5"]
>>> print(sum(nums), max(nums), min(nums))
Traceback (most recent call last):
  File "/home/DCI/test.py", line 30, in <module>
    print(sum(nums), max(nums), min(nums))
TypeError: unsupported operand type(s) for +:
'int' and 'str'
>>> nums = ["a", "b", "c", "d", "e"]
>>> print(max(nums), min(nums))
e a
```

Functions with Iterables: Sorted

Digital Career Institute

The **sorted** function returns a new list in alphabetical or numerical order.

It does not change the existing iterable.

Sorting a dictionary with **sorted** will return a list with the keys in alphabetical order, because the default iterable of a dictionary is by key.

Getting the **sorted** list of values can be done using the **values** method.

It accepts a keyword argument named **reverse** as a boolean that defaults to **False**.

```
>>> nums = [4, 3, 5, 2, 1]
>>> print(sorted(nums))
[1, 2, 3, 4, 5]

>>> print(nums)
[4, 3, 5, 2, 1]

>>> dict1 = {"c": 2, "b": 1, "a": 3}
>>> print(sorted(dict1))
['a', 'b', 'c']

>>> print(sorted(dict1.values()))
[1, 2, 3]

>>> print(sorted(dict1, reverse=True))
['c', 'b', 'a']
```

Functions with Iterables: Sorted

The **`sorted`** function can also be used on lists of dictionaries to sort the list based on the values of one of the keys in these dictionaries.

This can be done passing a function as the **`key`** argument. This function should return the value to be used for sorting.

```
>>> dict1 = [
...      {"name": "John", "age": 31},
...      {"name": "Mary", "age": 46},
...      {"name": "Lucy", "age": 25}
... ]
...
>>> by_age = lambda user: user["age"]
>>> by_name = lambda user: user["name"]

>>> print(sorted(dict1, key=by_age))
[{'name': 'Lucy', 'age': 25}, {'name': 'John',
'age': 31}, {'name': 'Mary', 'age': 46}]

>>> print(sorted(dict1, key=by_name))
[{'name': 'John', 'age': 31}, {'name': 'Lucy',
'age': 25}, {'name': 'Mary', 'age': 46}]
```

Functions with Iterables: Any & All

Some functions use iterables to return a boolean value indicating if the iterable matches a certain condition.

The function `any` will return `True` only if **any** of the values in the iterable is *truthy*.

The function `all` will return `True` only if **all** the values in the iterable are *truthy*.

```
>>> a_list = [1, True, "Mary", {1, 2}]
>>> print(bool(a_list), any(a_list), all(a_list))
True True True

>>> a_list = [1, True, "Mary", {}]
>>> print(bool(a_list), any(a_list), all(a_list))
True True False

>>> a_list = [0, False, "", {}]
>>> print(bool(a_list), any(a_list), all(a_list))
True False False
```

Functions with Iterables: Any & All

Some functions use iterables to return a boolean value indicating if the iterable matches a certain condition.

The function `any` will return `True` only if **any** of the values in the iterable is *truthy*.

The function `all` will return `True` only if **all** the values in the iterable are *truthy*.

```
>>> a_list = [1, True, "Mary", {1, 2}]
>>> print(bool(a_list), any(a_list), all(a_list))
True True True

>>> a_list = [1, True, "Mary", {}]
>>> print(bool(a_list), any(a_list), all(a_list))
True True False

>>> a_list = [0, False, "", {}]
>>> print(bool(a_list), any(a_list), all(a_list))
True False False
```

Functions with Iterables: Map

The most common functions in functional programming require both an iterable and a function.

This is the case of the **`map`** function, that applies the given function to each element of the given iterable.

It returns a **`map object`** that is an iterable containing the output of that process.

```
>>> a_list = [1, 2, 3, 4, 5]
>>> by_two = lambda num: num * 2
>>> a_list_by_two = map(by_two, a_list)
>>> print(a_list_by_two)
<map object at 0x7f11957546d0>

>>> print(list(a_list_by_two))
[2, 4, 6, 8, 10]
```

Functions with Iterables: Filter

The **`filter`** function returns an iterable with the elements of the given iterable that match a condition defined in a function.

It returns a **`filter object`** that is an iterable containing the output of that process.

```
>>> nums = [1, 2, 3, 4, 5]
>>> is_odd = lambda num: (num % 2) != 0
>>> odds = filter(is_odd, nums)

>>> print(odds)
<filter object at 0x7fced01036d0>

>>> print(list(odds))
[1, 3, 5]
```

Iterable Functions: Summary

CONSTRUCTORS

- list()
- tuple()
- set()
- dict()

PACKING

- enumerate()
- zip()

REDUCE

- sum()
- max()
- min()

ORDER

- sorted()

BOOLEAN

- any()
- all()

FUNCTIONAL

- map()
- filter()

Many built-in modules also provide functions that use iterables, like `random.shuffle` or `functools.reduce`.

We learned ...

- That Python has an abstract type named **iterable** that represents an object that can store different objects inside, which can be provided one at a time.
- That there are many built-in utility functions that use iterables to do various operations.
- That some functions, like **enumerate** and **zip** provide packing and unpacking functionalities to produce new iterables.
- Some of these functions are used to obtain boolean values (**any**, **all**) or a reduced number or string (**sum**, **max**, **min**).
- That some functions use both an iterable and a function, like **sorted**, **map** and **filter**.

Digital Career Institute