Day 21: Dictionary Methods

Removing Items

pop() method

The `pop()` method removes the item with the specified key name.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

popitem() method

The `popitem()` method removes the last inserted item.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.popitem()
print(thisdict)
```

del Keyword

The `del` keyword removes the item with the specified key name.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict["model"]
print(thisdict)

# Also deletes the dictionary completely
del thisdict
print(thisdict)
# this will cause an error because "thisdict" no longer exists.
```

clear() method

The `clear()` method empties the dictionary.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.clear()
print(thisdict)
```

Loop Dictionaries

Print all key names in the dictionary, one by one.

```
for x in thisdict:
  print(x)

# keys() method
for x in thisdict.keys():
  print(x)
```

Print all values in the dictionary, one by one.

```
for x in thisdict:
  print(thisdict[x])

# values() method
for x in thisdict.values():
  print(x)
```

Loop through both keys and values, by using the items() method.

```
for x, y in thisdict.items():
  print(x, y)
```

Copy Dictionaries

You cannot copy a dictionary simply by typing `dict2 = dict1`, because: `dict2` will only be a *reference* to `dict1`, and changes made in `dict1` will automatically also be made in `dict2`.

copy() method

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = thisdict.copy()
print(mydict)
```

Another way to make a copy is to use the built-in function `dict()`.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = dict(thisdict)
print(mydict)
```

Nested Dictionaries

A dictionary can contain dictionaries, this is called nested dictionaries.

```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}

# another way
child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}

myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}
```

Access Items in Nested Dictionaries

```
print(myfamily["child2"]["name"])
```

Loop through Nested Dictionaries

```
for x, obj in myfamily.items():
  print(x)

  for y in obj:
    print(y + ':', obj[y])
```

Lambda Function

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

Syntax:

```
lambda arguments : expression
```

```
x = lambda a : a + 10
print(x(5))
```

```
# output: 15
```

Lambda functions can take any number of arguments.

```
x = lambda a, b : a * b
print(x(5, 6))
# output: 30

# Example 2
x = lambda a, b, c: (a // b) + c
print(x(60, 10, 2))
# output: 8
```

Why use lambda functions?

The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number.

```
def myfunc(n):
  return lambda a : a * n



mydoubler = myfunc(2)
print(mydoubler(11))

mytripler = myfunc(3)
print(mytripler(11))
```

Note: Use lambda functions when an anonymous function is required for a short period of time.

Upcoming Classes

- Python Classes & Objects