Lecture 3

Python Dictionaries

Dictionaries are used to store data values in key:value pairs. A dictionary is a collection which is ordered, changeable and do not allow duplicates.

Dictionaries are written with curly brackets, and have keys and values:

In [1]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Dictionary Items

Dictionary items are ordered, changeable, and does not allow duplicates. Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

In [2]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict["brand"])
```

```
Ford
```

Dictionary Length

To determine how many items a dictionary has, use the len() function:

In [3]:
```
print(len(thisdict))
```

```
3
```

Dictionary Items - Data Types

The values in dictionary items can be of any data type:

In [4]:
```
thisdict = {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}
```

The dict() Constructor

It is also possible to use the dict() constructor to make a dictionary.

In [5]:
```
thisdict = dict(name = "John", age = 36, country = "Norway")
print(thisdict)
```

```
{'name': 'John', 'age': 36, 'country': 'Norway'}
```

Accessing Items

You can access the items of a dictionary by referring to its key name, inside square brackets:

In [6]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict["model"]
```

There is also a method called get() that will give you the same result:

In [7]:
```
x = thisdict.get("model")
```

Get Keys

The keys() method will return a list of all the keys in the dictionary.

In [8]:
```
x = thisdict.keys()
print(x)
```

```
dict_keys(['brand', 'model', 'year'])
```

Get Values

The values() method will return a list of all the values in the dictionary.

In [9]:
```
x = thisdict.values()
print(x)
```

```
dict_values(['Ford', 'Mustang', 1964])
```

Get Items

The items() method will return each item in a dictionary, as tuples in a list.

In [10]:
```
x = thisdict.items()
print(x)
```

```
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
```

Check if Key Exists

To determine if a specified key is present in a dictionary use the in keyword:

In [11]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

```
Yes, 'model' is one of the keys in the thisdict dictionary
```

Change Values

You can change the value of a specific item by referring to its key name (also used to add new items to the dictionary):

In [12]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"] = 2018
```

Update Dictionary

The update() method will update the dictionary with the items from the given argument. The argument must be a dictionary, or an iterable object with key:value pairs (also used to add new items to the dictionary).

In [19]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"year": 2020})
```

Removing Items

There are several methods to remove items from a dictionary:

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

In [14]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

```
thisdict.pop("model")
print(thisdict)
```

```
{'brand': 'Ford', 'year': 1964}
```

The popitem() method removes the last inserted item (in versions before 3.7, a random item is removed instead):

In [15]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.popitem()
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang'}
```

The del keyword removes the item with the specified key name:

In [16]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict["model"]
print(thisdict)
```

```
{'brand': 'Ford', 'year': 1964}
```

The clear() method empties the dictionary:

In [17]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.clear()
print(thisdict)
```

```
{}
```

Loop Through a Dictionary

You can loop through a dictionary by using a for loop. When looping through a dictionary, the return value are the keys of the dictionary, but there are methods to return the values as well.

In [20]:
```
for x in thisdict:
  print(x)
```

```
brand
model
year
```

In [21]:
```
for x in thisdict:
  print(thisdict[x])
```

```
Ford
Mustang
2020
```

You can also use the values() method to return values of a dictionary:

In [23]:
```
for x in thisdict.values():
  print(x)
```

```
Ford
Mustang
2020
```

You can use the keys() method to return the keys of a dictionary:

In [24]:
```
for x in thisdict.keys():
  print(x)
```

```
brand
model
year
```

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

In [25]:
```
for x, y in thisdict.items():
  print(x, y)
```

```
brand Ford
model Mustang
year 2020
```

Copy a Dictionary

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.

There are ways to make a copy, one way is to use the built-in Dictionary method copy().

In [26]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = thisdict.copy()
print(mydict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

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

In [27]:
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = dict(thisdict)
print(mydict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Nested Dictionaries

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

In [28]:
```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}
```

Or, if you want to add three dictionaries into a new dictionary:

In [29]:
```
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

To access items from a nested dictionary, you use the name of the dictionaries, starting with the outer dictionary:

In [30]:
```
print(myfamily["child2"]["name"])
```

```
Tobias
```

Dictionary Methods

Python has a set of built-in methods that you can use on dictionaries.

- clear(): Removes all the elements from the dictionary
- copy(): Returns a copy of the dictionary
- fromkeys(): Returns a dictionary with the specified keys and value
- get(): Returns the value of the specified key
- items(): Returns a list containing a tuple for each key value pair
- keys(): Returns a list containing the dictionary's keys
- pop(): Removes the element with the specified key
- popitem(): Removes the last inserted key-value pair
- setdefault(): Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
- update(): Updates the dictionary with the specified key-value pairs
- values(): Returns a list of all the values in the dictionary

Python Functions

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

Creating a Function

In Python a function is defined using the def keyword:

In [31]:
```python
def my_function():
  print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

In [32]:
```python
my_function()
```

```
Hello from a function
```

Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

In [33]:
```python
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

```
Emil Refsnes
Tobias Refsnes
Linus Refsnes
```

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

In [34]:
```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil", "Refsnes")
```

```
Emil Refsnes
```

Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a tuple of arguments, and can access the items accordingly:

In [37]:
```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

```
The youngest child is Linus
```

Keyword Arguments

You can also send arguments with the key = value syntax.

This way the order of the arguments does not matter.

In [38]:
```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

```
The youngest child is Linus
```

Arbitrary Keyword Arguments, **kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.

This way the function will receive a dictionary of arguments, and can access the items accordingly:

In [39]:
```
def my_function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")
```

```
His last name is Refsnes
```

Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without argument, it uses the default value:

In [40]:
```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

```
I am from Sweden
I am from India
I am from Norway
I am from Brazil
```

Return Values

To let a function return a value, use the return statement:

In [41]:
```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

```
15
25
45
```

The pass Statement

function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

In [42]:
```
def myfunction():
  pass
```

Recursion

Python also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

In [43]:
```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)

print(factorial(5))
```

```
120
```

Python Lambda

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

The expression is executed and the result is returned:

In [44]:
```
x = lambda a : a + 10
print(x(5))
```

```
15
```

Lambda functions can take any number of arguments:

In [45]:
```
x = lambda a, b : a * b
print(x(5, 6))
```

```
30
```

In [46]:
```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

```
13
```

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:

In [47]:
```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

```
22
33
```

In [ ]: