# Business Intelligence-relaterade programspråk

## OO, Math

# Lambda

- A lambda function is a small anonymous function.
- Use lambda functions when an anonymous function is required for a short period of time.
- A lambda function can take any number of arguments, but can only have one expression.

```
lambda arguments : expression
```

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

- Multiply argument a with argument b and return the result:

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

- Summarize argument a, b, and c and return the result:

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

# 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
```

- Use that function definition to make a function that always doubles the number you send in:

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

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

- Or, use the same function definition to make a function that always triples the number you send in:

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

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

# Why Use Lambda Functions?

- Or, use the same function definition to make both functions, in the same program:

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

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

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

# Avancerade variabler

- Saker i riktiga världen består ofta att olika variabler.
- Hänger de ihop på något sätt?

```
name = "Micke";
age = 44;
shoe_size = 43;
```

# Objekt & klasser

- Ett objekt är i sin enklaste form en variabel som har under-variabler.
- En klass är lite som en ritning eller mall för ett objekt.

# Classes and Objects

- Python is an object oriented programming language.
- Almost everything in Python is an object, with its properties and methods.
- A Class is like an object constructor, or a "blueprint" for creating objects.

# Create a Class and an Object

- Create a class named MyClass, with a property named x:

```
class MyClass:
  x = 5
```

- Klassen kan nu användas som mall för objekt.
- Create an object named p1, and print the value of x:

```
p1 = MyClass()
print(p1.x)   # 5
```

# The __init__() Function

- The examples above are classes and objects in their simplest form, and are not really useful in real life applications.
- To understand the meaning of classes we have to understand the built-in `__init__()` function.
- All classes have a function called `__init__()`, which is always executed when the class is being initiated.
- Use the `__init__()` function to assign values to object properties, or other operations that are necessary to do when the object is being created.

# The __init__() Function

- `__init__()` is **not** a constructor.
- `__init__()` is called immediately after the object is created and is used to initialize it.
- The constructor is called `__new__ ()`.
- https://medium.com/analytics-vidhya/is-init-the-constructor-in-python-43bb744e73b

# The __init__() Function

- Create a class named Person, use the `__init__()` function to assign values for name and age:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

# Exercise

- Create a class named `Vehicle`, use the `__init__()` function to assign values for `make`, `model`, `color`, `year`, `velocity` and `no_of_wheels`.
- `Velocity` should always be set to `0`.
- Let `no_of_wheels` be optional when creating an object by setting the default value `4` for the appropriate `__init__()` parameter.
- Create a couple of vehicles and print out their properties.

# Inheritance

- Inheritance allows us to define a class that inherits all the methods and properties from another class.
- *Parent class* is the class being inherited from, also called *base class* or *super class*.
- *Child class* is the class that inherits from another class, also called *derived class* or *sub class*.

# Create a Parent Class

- Any class can be a parent class, so the syntax is the same as creating any other class.
- Create a class named Person, with `firstname` and `lastname` properties, and a `printname` method:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

# Use the Person class to create an object, and then execute the printname method:

x = Person("John", "Doe")
x.printname()
```

# Create a Child Class

- To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class.
- Create a class named Student, which will inherit the properties and methods from the Person class.

```
class Student(Person):
  pass
```

- *Note:* Use the pass keyword when you do not want to add any other properties or methods to the class.

- Now the Student class has the same properties and methods as the Person class.
- Use the Student class to create an object, and then execute the `printname` method:

```
x = Student("Mike", "Olsen")
x.printname()
```

# Add the __init__() Function

- So far we have created a child class that inherits the properties and methods from its parent.
- We want to add the `__init__()` function to the child class (instead of the pass keyword).
- *Note:* The `__init__()` function is called automatically every time the class is being used to create a new object.

- Add the `__init__()` function to the Student class.

```
class Student(Person):
  def __init__(self, fname, lname):
    #add properties etc.
```

- When you add the `__init__()` function, the child class will no longer inherit the parent's `__init__()` function.
- Note: The child's `__init__()` function overrides the inheritance of the parent's `__init__()` function.

# Add the __init__() Function

- To keep the inheritance of the parent's `__init__()` function, add a call to the parent's `__init__()` function.

```
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
```

- Now we have successfully added the `__init__()` function, and kept the inheritance of the parent class, and we are ready to add functionality in the `__init__()` function.

# Use the super() Function

- Python also has a `super()` function that will make the child class inherit all the methods and properties from its parent.

```
def __init__(self, fname, lname):
  super().__init__(fname, lname)
```

- By using the `super()` function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

# Add Properties

- Add a property called `graduationyear` to the Student class.

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019
```

- In the example below, the year 2019 should be a variable, and passed into the Student class when creating student objects. To do so, add another parameter in the `__init__()` function:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

x = Student("Mike", "Olsen", 2019)
```

# Methods

- Objects have properties, ”sub variables”.
- Objects can also have functions.
  - Functions in classes are called *methods*.

# Add Methods

- Add a method called welcome to the Student class.

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class of",
self.graduationyear)
```

- If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

# Exercise

- Add the max_speed property to the Vehicle class.
- Create the methods `accelerate()` and `decelerate()` that both take a change parameter and increases / decreases the velocity.
  - Check the parameter. The new speed can’t be lower than 0 or higher than the max speed.
- Create a class named Car and let it inherit from Vehicle. The max_speed should always be 160.
- Create a class named Truck and let it inherit from Vehicle.
  - The max_speed should always be 80.
  - Add the property load_capacity.
- Create a couple of cars and trucks and print out their properties.
- Create the method print() that prints out the properties for the vehicle.
- Create the class Motorcycle that inherits from Vehicle. Max speed is 200.

# Iterators

- An iterator is an object that contains a countable number of values.
- An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.
- Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods `__iter__()` and `__next__()`.

# Iterator vs Iterable

- Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable containers which you can get an iterator from.
- All these objects have a `iter()` method which is used to get an *iterator*.
- Return an iterator from a tuple, and print each value:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))
```

# Iterator vs Iterable

- Even strings are iterable objects, and can return an iterator.
- Strings are also iterable objects, containing a sequence of characters.

```
mystr = "banana"
myit = iter(mystr)

print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

# Looping Through an Iterator

- We can use a `for` loop to iterate through an iterable object.
- Iterate the values of a tuple:

```
mytuple = ("apple", "banana", "cherry")

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

- The `for` loop actually creates an iterator object and executes the `next()` method for each loop.

# Create an Iterator

- To create an object/class as an iterator you have to implement the methods `__iter__()` and `__next__()` to your object.
- As you have learned in the Python Classes/Objects chapter, all classes have a function called `__init__()`, which allows you to do some initializing when the object is being created.
- The `__iter__()` method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.
- The `__next__()` method also allows you to do operations, and must return the next item in the sequence.

# Create an Iterator

- Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.)

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    x = self.a
    self.a += 1
    return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

# StopIteration

- The example above would continue forever if you had enough `next()` statements, or if it was used in a `for` loop.
- To prevent the iteration to go on forever, we can use the `StopIteration` statement.
- In the `__next__()` method, we can add a terminating condition to raise an error if the iteration is done a specified number of times

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

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

# Exercise

- Write an iterator class `reverse_iter`, that takes a list and iterates it from the reverse direction.

```
it = reverse_iter([1, 2, 3, 4])

next(it)   # 4
next(it)   # 3
next(it)   # 2
next(it)   # 1
next(it)   # StopIteration
```

# Generators

- Generators simplifies creation of iterators.
- A generator is a function that produces a sequence of results instead of a single value.

```
def yrange(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

- Each time the yield statement is executed the function generates a new value.

```
y = yrange(3)
next(y)   # 0
next(y)   # 1
next(y)   # 2
next(y)   # StopIteration
```

# Generators

- So a generator is also an iterator. You don’t have to worry about the iterator protocol.
- The word “generator” is confusingly used to mean both the function that generates and what it generates. In this chapter, I’ll use the word “generator” to mean the genearted object and “generator function” to mean the function that generates it.
- When a generator function is called, it returns a generator object without even beginning execution of the function. When next method is called for the first time, the function starts executing until it reaches yield statement. The yielded value is returned by the next call.

# Generators

- The following example demonstrates the interplay between `yield` and call to `__next__` method on generator object.

```
def integers():
    """Infinite sequence of integers."""
    i = 1
    while True:
        yield i
        i = i + 1
```

```
def squares():
    for i in integers():
        yield i * i

def take(n, seq):
    """Returns first n values from the given
sequence."""
    seq = iter(seq)
    result = []
    try:
        for i in range(n):
            result.append(next(seq))
    except StopIteration:
        pass
    return result

print(take(5, squares())) # prints [1, 4, 9, 16,
25]
```

# Modules

- What is a Module?
  - Consider a module to be the same as a code library.
  - A file containing a set of functions you want to include in your application.

# Create a Module

- To create a module just save the code you want in a file with the file extension `.py`.

```
def greeting(name):
  print("Hello, " + name)
```

# Use a Module

- Now we can use the module we just created, by using the import statement.

```
import mymodule
mymodule.greeting(”Jonathan”)
```

- Note: When using a function from a module, use the syntax: `module_name.function_name`.

# Naming a Module

- You can name the module file whatever you like, but it must have the file extension `.py`
- Re-naming a Module
  - You can create an alias when you import a module, by using the `as` keyword:

```
import mymodule as mx

a = mx.person1["age"]
print(a)
```

# Built-in Modules

- There are several built-in modules in Python, which you can import whenever you like.
- Import and use the platform module

```
import platform

x = platform.system()
print(x)
```

# Using the dir() Function

- There is a built-in function to list all the function names (or variable names) in a module, the `dir()` function.
- List all the defined names belonging to the platform module:

```
import platform

x = dir(platform)
print(x)
```

- *Note:* The `dir()` function can be used on all modules, also the ones you create yourself.

# Import From Module

- You can choose to import only parts from a module, by using the `from` keyword.
- The module named `mymodule` has one function and one dictionary:

```
def greeting(name):
  print("Hello, " + name)

person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

- Import only the person1 dictionary from the module:

```
from mymodule import person1

print (person1["age"])
```

- Note: When importing using the from keyword, do not use the module name when referring to elements in the module. Example: `person1["age"]`, **not** ~~`mymodule.person1["age"]`~~

# Datetime

- A date in Python is not a data type of its own, but we can import a module named `datetime` to work with dates as date objects.
- Import the `datetime` module and display the current date:

```
import datetime

x = datetime.datetime.now()
print(x)
```

# Date Output

- When we execute the code from the example above the result will be:

```
2021-03-30 03:26:10.505818
```

- The date contains year, month, day, hour, minute, second, and microsecond.
- The `datetime` module has many methods to return information about the date object.

# Date Output

- Return the year and name of weekday:

```
import datetime

x = datetime.datetime.now()

print(x.year)
print(x.strftime("%A"))
```

# Creating Date Objects

- To create a date, we can use the `datetime()` class (constructor) of the `datetime` module.
- The `datetime()` class requires three parameters to create a date: year, month, day.

```
import datetime

x = datetime.datetime(2020, 5, 17)

print(x)
```

- The datetime() class also takes parameters for time and timezone (hour, minute, second, microsecond, tzone), but they are optional, and has a default value of 0, (None for timezone).

# The strftime() Method

- The `datetime` object has a method for formatting date objects into readable strings.
- The method is called `strftime()`, and takes one parameter, `format`, to specify the format of the returned string:
- Display the name of the month:

```
import datetime

x = datetime.datetime(2018, 6, 1)

print(x.strftime("%B"))
```

# Date Format Reference

| Directive | Description | Example |
|---|---|---|
| %a | Weekday, short version | Wed |
| %A | Weekday, full version | Wednesday |
| %w | Weekday as a number 0-6, 0 is Sunday | 3 |
| %d | Day of month 01-31 | 31 |
| %b | Month name, short version | Dec |
| %B | Month name, full version | December |
| %m | Month as a number 01-12 | 12 |
| %y | Year, short version, without century | 18 |
| %Y | Year, full version | 2018 |
| %H | Hour 00-23 | 17 |
| %I | Hour 00-12 | 05 |
| %p | AM/PM | PM |
| %M | Minute 00-59 | 41 |
| %S | Second 00-59 | 08 |
| %f | Microsecond 000000-999999 | 548513 |
| %z | UTC offset | +0100 |
| %Z | Timezone | CST |
| %j | Day number of year 001-366 | 365 |
| %U | Week number of year, Sunday as the first day of week, 00-53 | 52 |
| %W | Week number of year, Monday as the first day of week, 00-53 | 52 |
| %c | Local version of date and time | Mon Dec 31 17:41:0 |
| %x | Local version of date | 12/31/18 |
| %X | Local version of time | 17:41:00 |
| %% | A % character | % |
| %G | ISO 8601 year | 2018 |
| %u | ISO 8601 weekday (1-7) | 1 |
| %V | ISO 8601 weeknumber (01-53) | 01 |

# Exercise

- Write a program that takes an input date/time from a user and outputs the following formats.
    1. Current date and time
    2. Current year
    3. Month of year
    4. Week number of the year
    5. Weekday of the week
    6. Day of year
    7. Day of the month
    8. Day of week

# Math

- Python has a set of built-in math functions, including an extensive math module, that allows you to perform mathematical tasks on numbers.
- Some built in functions:
  - The `min()` and `max()` functions can be used to find the lowest or highest value in an iterable:
  - x = min(5, 10, 25)
  - y = max(5, 10, 25)

# Math

- The `abs()` function returns the absolute (positive) value of the specified number:

```
x = abs(-7.25)
print(x)
```

- The `pow(x, y)` function returns the value of `x` to the power of `y`.
- Return the value of 4 to the power of 3 (same as 4 * 4 * 4):

```
x = pow(4, 3)
print(x)
```

# The Math Module

- Python has also a built-in module called math, which extends the list of mathematical functions.
- When you have imported the math module, you can start using methods and constants of the module.
- The `math.sqrt()` method for example, returns the square root of a number:

```
import math

x = math.sqrt(64)

print(x)
```

- The math.pi constant, returns the value of PI (3.14...)

# The Math Module

- The `math.ceil()` method rounds a number upwards to its nearest integer, and the `math.floor()` method rounds a number downwards to its nearest integer, and returns the result.

```
import math

x = math.ceil(1.4)
y = math.floor(1.4)

print(x) # returns 2
print(y) # returns 1
```

# The Math Module

- Read more in the documentation.
- https://docs.python.org/3/library/math.html

# The Math Module

- Read more in the documentation.
- https://docs.python.org/3/library/math.html

# Exercise

- Write the class `Rectangle` with properties `x` and `y`.
  - Add methods `area()` and `circumference()`.
- Write the class `Square` that inherits from `Rectangle`.
- Write the class `Circle` that uses the `pi` constant.
- Write the class `Triangle` that uses different math methods like `tan`, `sin` and `cos` etc to calculate area and circumference depending on what sides/angles you have access to.

# JSON

- JSON is a syntax for storing and exchanging data.
- JSON is text, written with JavaScript object notation.
- Python has a built-in package called json, which can be used to work with JSON data.

```
import json
```

# Parse JSON - Convert from JSON to Python

- If you have a JSON string, you can parse it by using the `json.loads()` method.
- The result will be a Python dictionary.

```
import json

# some JSON:
x =  '{ "name":"John", "age":30, "city":"New York"}'

# parse x:
y = json.loads(x)

# the result is a Python dictionary:
print(y["age"])
```

# Convert from Python to JSON

- If you have a Python object, you can convert it into a JSON string by using the `json.dumps()` method.

```
import json

# a Python object (dict):
x = {
  "name": "John",
  "age": 30,
  "city": "New York"
}

# convert into JSON:
y = json.dumps(x)

# the result is a JSON string:
print(y)
```

# Convert from Python to JSON

- You can convert Python objects of the following types, into JSON strings:
- dict
- list
- tuple
- string
- int
- float
- True
- False
- None

- Convert Python objects into JSON strings, and print the values:

```
import json

print(json.dumps({"name": "John", "age": 30}))
print(json.dumps(["apple", "bananas"]))
print(json.dumps(("apple", "bananas")))
print(json.dumps("hello"))
print(json.dumps(42))
print(json.dumps(31.76))
print(json.dumps(True))
print(json.dumps(False))
print(json.dumps(None))
```

# Convert from Python to JSON

- When you convert from Python to JSON, Python objects are converted into the JSON (JavaScript) equivalent.

| Python | JSON |
|---|---|
| dict | Object |
| list | Array |
| tuple | Array |
| str | String |
| int | Number |
| float | Number |
| True | true |
| False | false |
| None | null |

# Convert from Python to JSON

## Convert a Python object containing all the legal data types

```
import json

x = {
  "name": "John",
  "age": 30,
  "married": True,
  "divorced": False,
  "children": ("Ann","Billy"),
  "pets": None,
  "cars": [
    {"model": "BMW 230", "mpg": 27.5},
    {"model": "Ford Edge", "mpg": 24.1}
  ]
}

print(json.dumps(x))
```

# Convert from Python to JSON

## Format the Result

- The example above prints a JSON string, but it is not very easy to read, with no indentations and line breaks.
- The `json.dumps()` method has parameters to make it easier to read the result:
- Use the indent parameter to define the numbers of indents:

```
json.dumps(x, indent=4)
```

- You can also define the separators, default value is (", ", ": "), which means using a comma and a space to separate each object, and a colon and a space to separate keys from values:
- Use the separators parameter to change the default separator:

```
json.dumps(x, indent=4, separators=(". ", " = "))
```

# Convert from Python to JSON

## Order the Result

- The `json.dumps()` method has parameters to order the keys in the result:
- Use the `sort_keys` parameter to specify if the result should be sorted or not:

```
json.dumps(x, indent=4, sort_keys=True)
```

# RegEx

- A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.
- RegEx can be used to check if a string contains the specified search pattern.
- Python has a built-in package called `re`, which can be used to work with Regular Expressions.

```
import re
```

# RegEx

- When you have imported the `re` module, you can start using regular expressions.
- Search the string to see if it starts with "The" and ends with "Spain":

```
import re

txt = "The rain in Spain"
x = re.search("^The.*Spain$", txt)
```

# RegEx Functions

- The `re` module offers a set of functions that allows us to search a string for a match.

| Function | Description |
|---|---|
| findall | Returns a list containing all matches |
| search | Returns a Match object if there is a match anywhere in the string |
| split | Returns a list where the string has been split at each match |
| sub | Replaces one or many matches with a string |

# RegEx

## Metacharacters

| Character | Description | Example |
|---|---|---|
| [] | A set of characters | "[a-m]" |
| \ | Signals a special sequence (can also be used to escape special characters) | "\d" |
| . | Any character (except newline character) | "he..o" |
| ^ | Starts with | "^hello" |
| $ | Ends with | "world$" |
| * | Zero or more occurrences | "aix*" |
| + | One or more occurrences | "aix+" |
| {} | Exactly the specified number of occurrences | "al{2}" |
| \| | Either or | "falls\|stays" |
| () | Capture and group | |

# RegEx

## Special Sequences

| Character | Description | Example |
|---|---|---|
| \A | Returns a match if the specified characters are at the beginning of the string | "\AThe" |
| \b | Returns a match where the specified characters are at the beginning or at the end of a word<br>(the "r" in the beginning is making sure that the string is being treated as a "raw string") | r"\bain"<br>r"ain\b" |
| \B | Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word<br>(the "r" in the beginning is making sure that the string is being treated as a "raw string") | r"\Bain"<br>r"ain\B" |
| \d | Returns a match where the string contains digits (numbers from 0-9) | "\d" |
| \D | Returns a match where the string DOES NOT contain digits | "\D" |
| \s | Returns a match where the string contains a white space character | "\s" |
| \S | Returns a match where the string DOES NOT contain a white space character | "\S" |
| \w | Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character) | "\w" |
| \W | Returns a match where the string DOES NOT contain any word characters | "\W" |
| \Z | Returns a match if the specified characters are at the end of the string | "Spain\Z" |

# RegEx

## Sets

| Set | Description |
|---|---|
| [arn] | Returns a match where one of the specified characters ( `a` , `r` , or `n` ) are present |
| [a-n] | Returns a match for any lower case character, alphabetically between `a` and `n` |
| [^arn] | Returns a match for any character EXCEPT `a` , `r` , and `n` |
| [0123] | Returns a match where any of the specified digits ( `0` , `1` , `2` , or `3` ) are present |
| [0-9] | Returns a match for any digit between `0` and `9` |
| [0-5][0-9] | Returns a match for any two-digit numbers from `00` and `59` |
| [a-zA-Z] | Returns a match for any character alphabetically between `a` and `z` , lower case OR upper case |
| [+] | In sets, `+` , `*` , `.` , `\|` , `()` , `$` , `{}` has no special meaning, so `[+]` means: return a match for any `+` character in the string |

# RegEx

## The findall() Function

- The `findall()` function returns a list containing all matches.

```
import re

txt = "The rain in Spain"
x = re.findall("ai", txt)
print(x)
```

- The list contains the matches in the order they are found.
- If no matches are found, an empty list is returned.

```
import re

txt = "The rain in Spain"
x = re.findall("Portugal", txt)
print(x)
```

# RegEx

## The search() Function

- The `search()` function searches the string for a match, and returns a Match object if there is a match.
- If there is more than one match, only the first occurrence of the match will be returned:
- Search for the first white-space character in the string:

```
import re

txt = "The rain in Spain"
x = re.search("\s", txt)

print("The first white-space character is
located in position:", x.start())
```

- If no matches are found, the value None is returned.

```
import re

txt = "The rain in Spain"
x = re.search("Portugal", txt)
print(x)
```

# RegEx

## The split() Function

- The `split()` function returns a list where the string has been split at each match.
- Split at each white-space character:

```
import re

txt = "The rain in Spain"
x = re.split("\s", txt)
print(x)
```

- You can control the number of occurrences by specifying the maxsplit parameter.
- Split the string only at the first occurrence:

```
import re

txt = "The rain in Spain"
x = re.split("\s", txt, 1)
print(x)
```

# RegEx

## The sub() Function

- The `sub()` function replaces the matches with the text of your choice:
- Replace every white-space character with the number 9:

```
import re

txt = "The rain in Spain"
x = re.sub("\s", "9", txt)
print(x)
```

- You can control the number of replacements by specifying the count parameter:
- Replace the first 2 occurrences:

```
import re

txt = "The rain in Spain"
x = re.sub("\s", "9", txt, 2)
print(x)
```

# RegEx

## Match Object

- A Match Object is an object containing information about the search and the result.
- *Note:* If there is no match, the value None will be returned, instead of the Match Object.
- Do a search that will return a Match Object:

```
import re

txt = "The rain in Spain"
x = re.search("ai", txt)
print(x) #this will print an object
```

# RegEx
## Match Object

- The Match object has properties and methods used to retrieve information about the search, and the result.

- `.span()` returns a tuple containing the start-, and end positions of the match.
  `.string` returns the string passed into the function
  `.group()` returns the part of the string where there was a match

- Print the position (start- and end-position) of the first match occurrence.

- The regular expression looks for any words that starts with an upper case "S":

```
import re

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.span())
```

# RegEx

## Match Object

- Print the string passed into the function:

```
import re

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.string)
```

- Print the part of the string where there was a match.

- The regular expression looks for any words that starts with an upper case "S":

```
import re

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.group())
```

- *Note:* If there is no match, the value None will be returned, instead of the Match Object.

# Exercise

- Try out some of the RegEx exercises:
- https://www.w3resource.com/python-exercises/re/index.php
- Feel free to check out the solutions, especially for you first couple of exercises, but try to avoid them after that unless you get stuck. Remember that we want to be on the third step on the programming ladder, not the first one.

# PIP

- PIP is a package manager for Python packages, or modules if you like.
- *Note:* If you have Python version 3.4 or later, PIP is included by default.
- A package contains all the files you need for a module.
- Modules are Python code libraries you can include in your project.
- Check PIP version:

```
C:\>pip --version
```

# PIP

## Download a Package

- Open the command line interface and tell PIP to download the package you want.
- Download a package named "camelcase":

```
C:\>pip install camelcase
```

- Now you have downloaded and installed your first package!

# PIP

## Using a Package

- Once the package is installed, it is ready to use.
- Import the "camelcase" package into your project.

```
import camelcase

c = camelcase.CamelCase()

txt = "hello world"

print(c.hump(txt))
```

# PIP

## Find Packages

- Find more packages at https://pypi.org/.

# PIP

## Remove a Package

- Use the uninstall command to remove a package:

```
C:\>pip uninstall camelcase
```

- The PIP Package Manager will ask you to confirm that you want to remove the camelcase package.

# PIP

## List Packages

- Use the `list` command to list all the packages installed on your system.

```
C:\>pip list
Package          Version
-----------------------
camelcase        0.2
mysql-connector 2.1.6
pip              18.1
pymongo          3.6.1
setuptools       39.0.1
```

# Try Except

- The `try` block lets you test a block of code for errors.
- The `except` block lets you handle the error.
- The `finally` block lets you execute code, regardless of the result of the try- and except blocks.

# Exception Handling

- When an error occurs, or exception as we call it, Python will normally stop and generate an error message.
- These exceptions can be handled using the try statement:
- The try block will generate an exception, because `x` is not defined:

```
try:
  print(x)
except:
  print("An exception occurred")
```

- Since the try block raises an error, the except block will be executed.
- Without the try block, the program will crash and raise an error.

# Many Exceptions

- You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:
- Print one message if the try block raises a NameError and another for other errors:

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

# Exceptions

## Else

- You can use the `else` keyword to define a block of code to be executed if no errors were raised:
- In this example, the `try` block does not generate any error:

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

# Exceptions

## Finally

- The finally block, if specified, will be executed regardless if the try block raises an error or not.

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

- This can be useful to close objects and clean up resources:

- Try to open and write to a file that is not writable:

```
try:
  f = open("demofile.txt")
  f.write("Lorum Ipsum")
except:
  print("Something went wrong when writing
to the file")
finally:
  f.close()
```

- The program can continue, without leaving the file object open.

# Raise an exception

- As a Python developer you can choose to throw an exception if a condition occurs.
- To throw (or raise) an exception, use the `raise` keyword.
- Raise an error and stop the program if x is lower than 0:

```
x = -1

if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

- The `raise` keyword is used to raise an exception.
- You can define what kind of error to raise, and the text to print to the user.
- Raise a TypeError if x is not an integer:

```
x = "hello"

if not type(x) is int:
  raise TypeError("Only integers are allowed")
```

# Exercise

## An interactive calculator

- You're going to write an interactive calculator! User input is assumed to be a formula that consist of a number, an operator (at least + and -), and another number, separated by white space (e.g. 1 + 1). Split user input using `str.split()`, and check whether the resulting list is valid:
- If the input does not consist of 3 elements, raise a `FormulaError`, which is a custom `Exception`.
- Try to convert the first and third input to a `float` (like so: `float_value = float(str_value)`). Catch any `ValueError` that occurs, and instead raise a `FormulaError`
- If the second input is not '+' or '-', again raise a FormulaError
- If the input is valid, perform the calculation and print out the result. The user is then prompted to provide new input, and so on, until the user enters `quit`.
- An interaction could look like this:

```
>>> 1 + 1
2.0
>>> 3.2 - 1.5
1.7000000000000002
>>> quit
```

- Solution: https://python.cogsci.nl/basic/exceptions-solution/

# User Input

- Python allows for user input.
- Python 3.6 uses the `input()` method.
- Python 2.7 uses the `raw_input()` method.
- The following example asks for the username, and when you entered the username, it gets printed on the screen:

- Python 3.6

```
username = input("Enter username:")
print("Username is: " + username)
```

# String Formatting

- To make sure a string will display as expected, we can format the result with the `format()` method.
- Sometimes there are parts of a text that you do not control, maybe they come from a database, or user input?
- To control such values, add placeholders (curly brackets `{}`) in the text, and run the values through the `format()` method:
- Add a placeholder where you want to display the price:

```
price = 49
txt = "The price is {} dollars"
print(txt.format(price))
```

# String Formatting

- You can add parameters inside the curly brackets to specify how to convert the value:
- Format the price to be displayed as a number with two decimals:

```
txt = "The price is {:.2f} dollars”
```

- https://docs.python.org/3/library/string.html

# String Formatting

## Multiple Values

- If you want to use more values, just add more values to the `format()` method.

```
print(txt.format(price, itemno, count))
```

- And add more placeholders.

```
quantity = 3
itemno = 567
price = 49
myorder = "I want {} pieces of item number {} for {:.2f} dollars."
print(myorder.format(quantity, itemno, price))
```

# String Formatting

## Index Numbers

- You can use index numbers (a number inside the curly brackets {0}) to be sure the values are placed in the correct placeholders:

```
quantity = 3
itemno = 567
price = 49
myorder = "I want {0} pieces of item number {1} for {2:.2f} dollars."
print(myorder.format(quantity, itemno, price))
```

- Also, if you want to refer to the same value more than once, use the index number.

```
age = 36
name = "John"
txt = "His name is {1}. {1} is {0} years old."
print(txt.format(age, name))
```

# String Formatting

## Named Indexes

- You can also use named indexes by entering a name inside the curly brackets `{carname}`, but then you must use names when you pass the parameter values `txt.format(carname = ”Ford")`.

```
myorder = "I have a {carname}, it is a {model}."
print(myorder.format(carname = "Ford", model = "Mustang"))
```

# File Handling

- File handling is an important part of any web application.
- Python has several functions for creating, reading, updating, and deleting files.
- The key function for working with files in Python is the `open()` function.
- The `open()` function takes two parameters; filename, and mode.
- There are four different methods (modes) for opening a file.
- `"r"` - Read - Default value. Opens a file for reading, error if the file does not exist
- `"a"` - Append - Opens a file for appending, creates the file if it does not exist
- `"w"` - Write - Opens a file for writing, creates the file if it does not exist
- `"x"` - Create - Creates the specified file, returns an error if the file exists
- In addition you can specify if the file should be handled as binary or text mode
- `"t"` - Text - Default value. Text mode
- `"b"` - Binary - Binary mode (e.g. images)

# File Handling

## Syntax

- To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

- The code above is the same as:

```
f = open("demofile.txt", "rt")
```

- Because `"r"` for read, and `"t"` for text are the default values, you do not need to specify them.

- *Note:* Make sure the file exists, or else you will get an error.

# File Handling

## Open a File on the Server

- Assume we have the following file, located in the same folder as Python:

```
# demofile.txt
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

- To open the file, use the built-in `open()` function.

- The `open()` function returns a file object, which has a `read()` method for reading the content of the file:

```
f = open("demofile.txt", "r")
print(f.read())
```

# File Handling

## Open a File on the Server

- If the file is located in a different location, you will have to specify the file path, like this.

```
f = open("D:\\myfiles\welcome.txt", "r")
print(f.read())
```

# File Handling

## Read Only Parts of the File

- By default the `read()` method returns the whole text, but you can also specify how many characters you want to return.
- Return the 5 first characters of the file:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

# File Handling

## Read lines

- You can return one line by using the `readline()` method:

```
f = open("demofile.txt", "r")
print(f.readline())
```

- By calling `readline()` two times, you can read the two first lines.

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

- By looping through the lines of the file, you can read the whole file, line by line:

- Loop through the file line by line:

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

# File Handling

## Close Files

- It is a good practice to always close the file when you are done with it.

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

- *Note:* You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

# File Handling

## Write to an Existing File

- To write to an existing file, you must add a parameter to the `open()` function.
- `"a"` - Append - will append to the end of the file
- `"w"` - Write - will overwrite any existing content
- Open the file "demofile2.txt" and append content to the file:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

- Open the file "demofile3.txt" and overwrite the content:

```
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()

#open and read the file after the appending:
f = open("demofile3.txt", "r")
print(f.read())
```

- *Note:* the `"w"` method will overwrite the entire file.

# File Handling

## Create a New File

- To create a new file in Python, use the `open()` method, with one of the following parameters.
- `"x"` - Create - will create a file, returns an error if the file exist
- `"a"` - Append - will create a file if the specified file does not exist
- `"w"` - Write - will create a file if the specified file does not exist

- Create a file called "myfile.txt":

```
f = open("myfile.txt", "x")
```

- Result: a new empty file is created!
- Create a new file if it does not exist:

```
f = open("myfile.txt", "w")
```

# File Handling

## Delete a File

- To delete a file, you must import the OS module, and run its `os.remove()` function:

- Remove the file "demofile.txt":

```
import os
os.remove("demofile.txt")
```

- To avoid getting an error, you might want to check if the file exists before you try to delete it.

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

# File Handling

## Delete Folder

- To delete an entire folder, use the `os.rmdir()` method.
- Remove the folder "myfolder":

```
import os
os.rmdir("myfolder")
```

- *Note:* You can only remove empty folders.

# File Handling

## List files

- `os.listdir()` will get you everything that's in a directory - files and directories.
- If you want just files, you could either filter this down using `os.path`:

```
from os import listdir
from os.path import isfile, join
onlyfiles = [f for f in listdir(mypath)
if isfile(join(mypath, f))]
```

- Or you could use `os.walk()` which will yield two lists for each directory it visits - splitting into files and dirs for you. If you only want the top directory you can break the first time it yields

```
from os import walk

f = []
for (dirpath, dirnames, filenames) in
walk(mypath):
    f.extend(filenames)
    break
```

# File Handling

- https://docs.python.org/3/library/string.html

# Exercise

- Do a couple of exercises here:
  - https://www.w3resource.com/python-exercises/file/index.php

# Python
## Övningar

- https://www.w3schools.com/python/exercise.asp
- https://www.w3resource.com/python-exercises/