

Exercises Python Programming

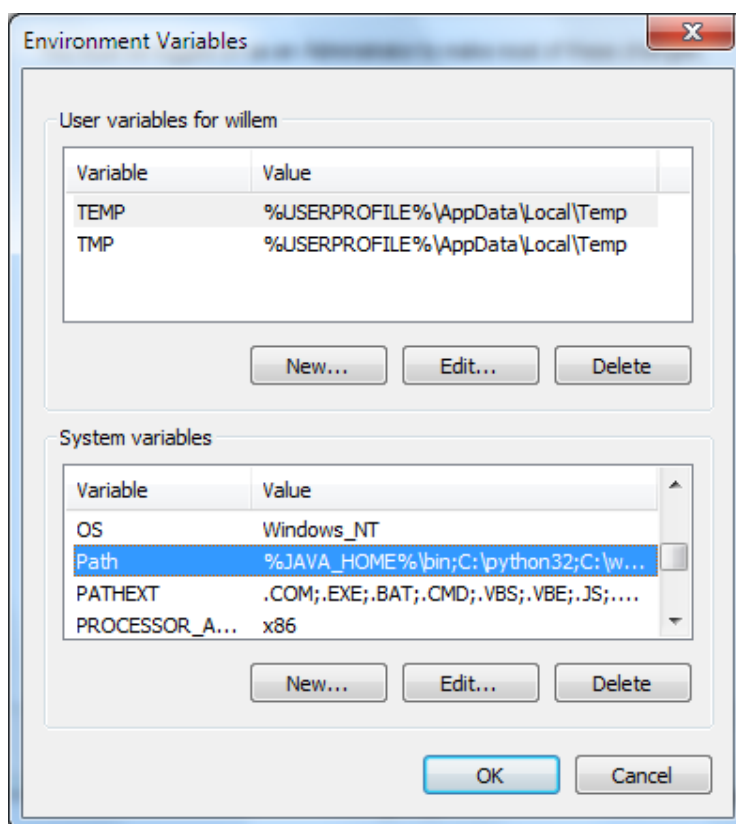
Exercise 0 : Environment Configuration	2
Exercise 1 : Python Intro	9
Exercise 2 : Variables and Data Types.....	10
Exercise 3 : Data Structures.....	12
Exercise 4 : Control Flow	15
Exercise 5 : Functions	17
Exercise 6 : Modules.....	18
Exercise 7 : Classes and Objects.....	19
Exercise 8 : Exception Handling.....	23
Exercise 9 : Python IO	24
Exercise 10 : Python Database Access.....	25
Exercise 11 : Python and XML	26
Exercise 12 : Python Libraries.....	27

Exercise 0 : Environment Configuration

We will use the Eclipse IDE with the Python Plugin Pydev for Python Development in this course. Most people know Eclipse as an integrated development environment (IDE) for Java. Eclipse is created by an open source community and is used in several other areas like PHP and Python development. This exercise covers the installation of Eclipse and the installation and configuration of the PyDev Plugin . This exercise is based on Eclipse Juno.

Step 1 . Python Installation

Double click `python-3.2.msi` from the software directory of your courseCD and do a full install of Python in `c:\Python32`. Go to Computer->Properties->Advanced System Settings->Environment Variables and put the directory `c:\python32` on the PATH.

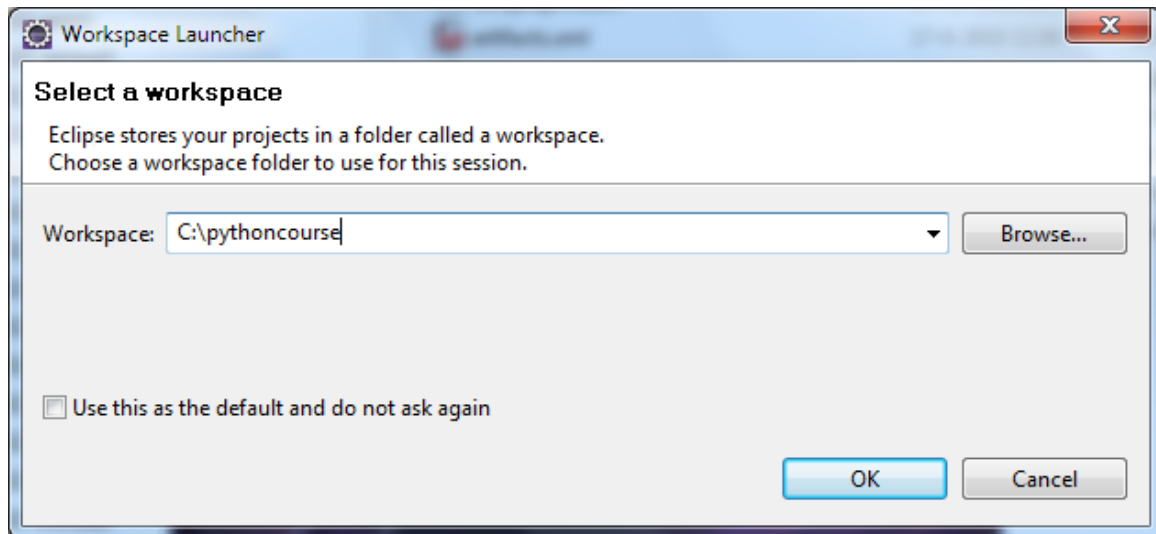


Step 2 . Eclipse Installation

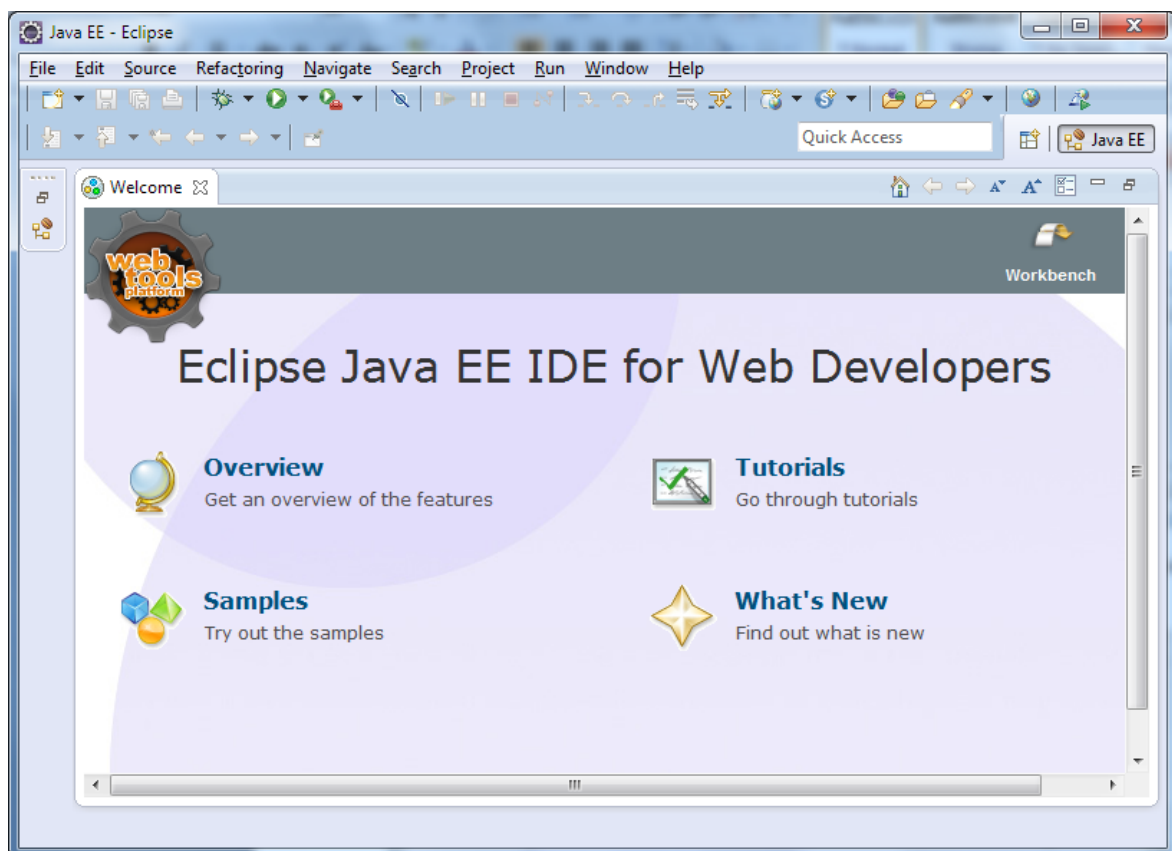
Eclipse requires an installed Java Runtime. The use of Java 7 is recommended. If necessary install the Java SDK from your course CD. Next download "Eclipse IDE for Java EE Developers" from the website Eclipse Downloads and unpack it to a directory. Alternatively you can use the file `eclipse-jee-juno-win32.zip` from your course CD and unpack it with the 7-zip utility. Install 7-zip first if necessary by double clicking `7z457.exe`. Use a directory path which does not contain spaces in its name as Eclipse sometimes has problems with that. After unpacking the download Eclipse is ready to be used; no additional installation procedure is required. The location `c:\eclipse` is recommended.

Step 3 . Start Eclipse

To start Eclipse double-click on the file "eclipse.exe". The system will prompt you for a workspace. The workspace is the place there you store your Java projects (more on workspaces later). Select an empty directory and press Ok. The recommended place for the workspace is `c:\pythoncourse`.



Eclipse will start and show the Welcome page. Close the welcome page by press the "X" besides the "Welcome".



Step 3 : Eclipse UI Overview

Eclipse provides perspectives, views and editors. Views and editors are grouped into perspectives. All projects are located in a workspace.

The workspace is the physical location (file path) you are working in. You can choose the workspace during startup of eclipse or via the menu (File-> Switch Workspace-> Others). All your projects, sources files, images and other artifacts will be stored and saved in your workspace.

You can predefine the workspace via the startup parameter `-data path_to_workspace`, e.g.

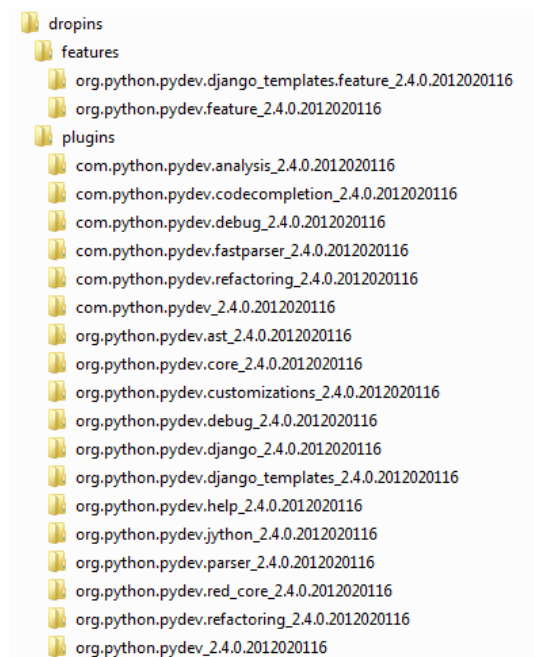
`"c:\eclipse.exe -data "c:\temp"` Please note that you have to put the path name into brackets. To see the current workspace directory in the title of Eclipse use `-showLocation` as additional parameter.

A perspective is a visual container for a set of views and editors. You can change the layout within a perspective (close / open views, editors, change the size, change the position, etc.). Eclipse allow you to switch to another perspective via the menu Window->Open Perspective -> Other. For Java development you usually use the "Java Perspective". A common problem is that you closed a view and don't know how to re-open this view. You can reset a perspective it to it original state via the menu "Window" -> "Reset Perspective". The default perspective in the "Eclipse IDE for Java EE Developers" is the Java EE perspective. Now change from the Java EE Perspective to the Java Perspective with Window->Open Perspective->Java

A view is typically used to navigate a hierarchy of information or to open an editor. Changes in a view are directly applied to the underlying data structure. Editors are used to modify elements. Editors can have code completion, undo / redo, etc. To apply the changes in an editor to the underlying resources, e.g. Java source file, you usually have to save.

Step 4 : Install PyDev Plugin

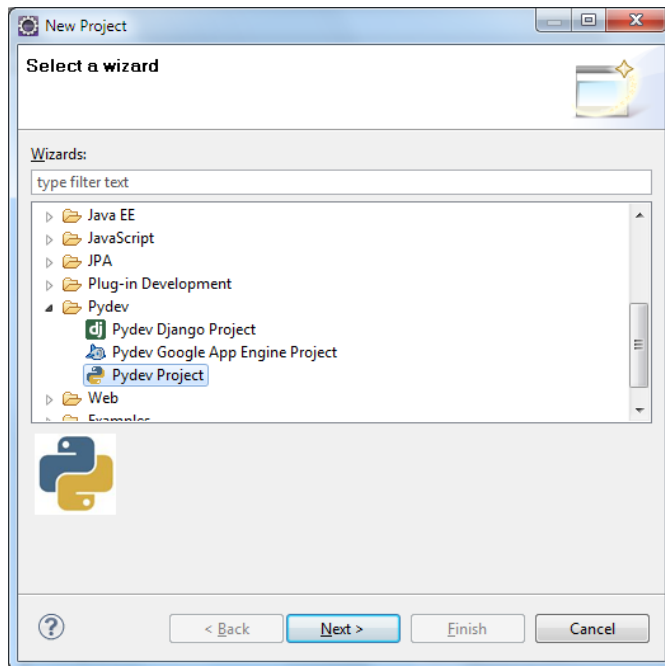
The PyDev plugin facilitates Python development in the Eclipse IDE. The plugin is available in the PyDev 2.4.0 directory in the software directory of your courseCD. Copy the directories `features` and `plugins` underneath the PyDev 2.4.0 directory to the `dropins` directory of your Eclipse installation directory which is typically `c:\eclipse`. The result should look as follows :



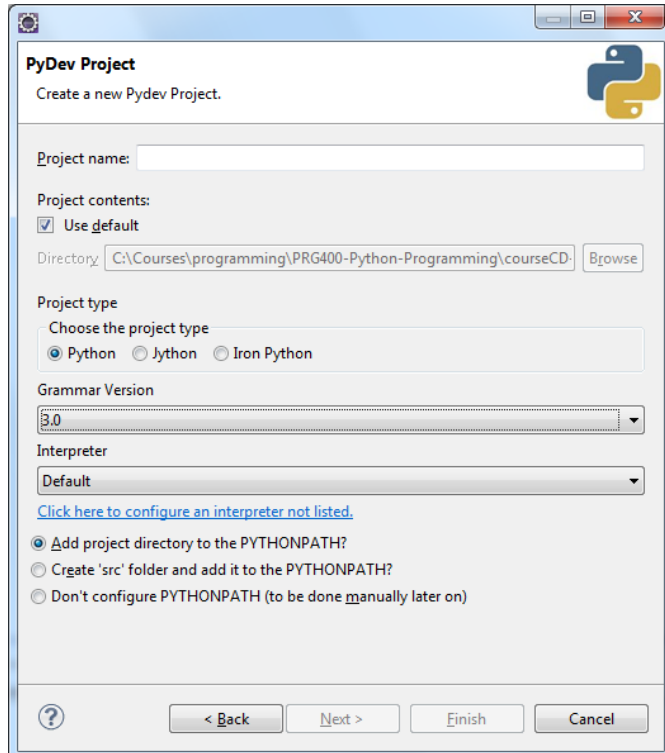
Now restart Eclipse to make the plugin available. You should see entries for PyDev under Window->Preferences.

Step 5 : Create a Python Project

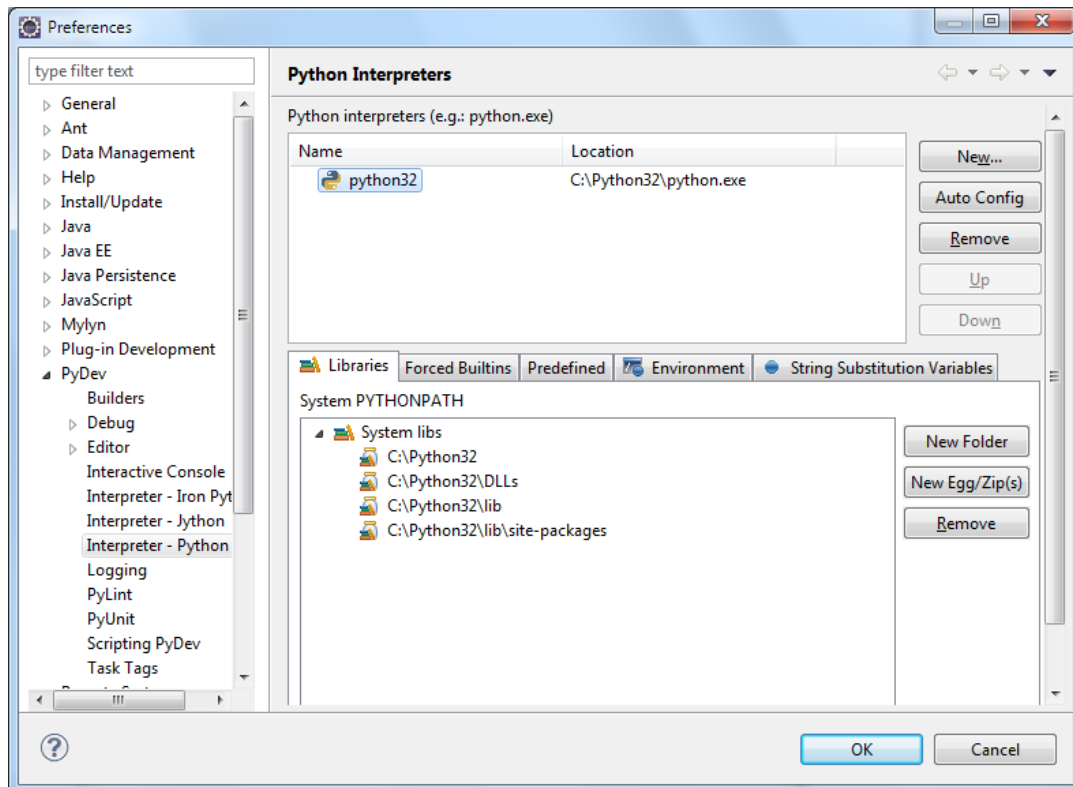
Create a new empty Python project by clicking File->New->Project->PyDev->PyDev Project.



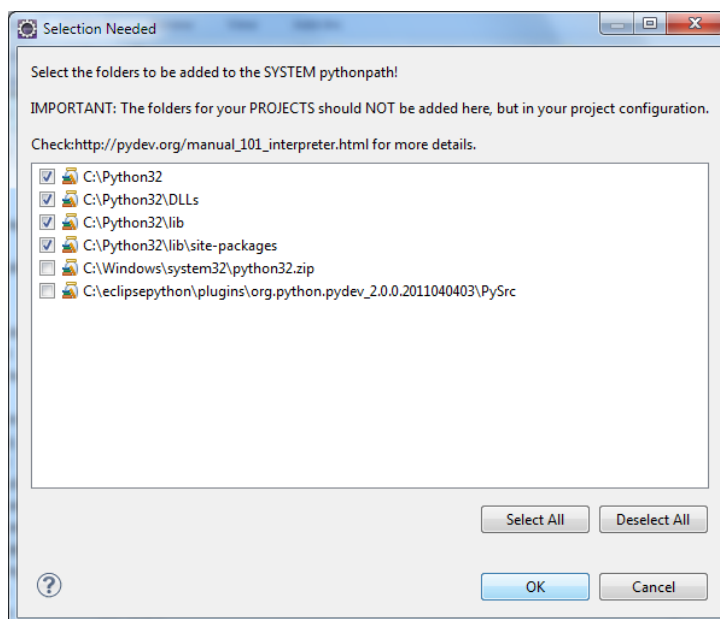
Click Next. Set the Grammar Version to 3.0.



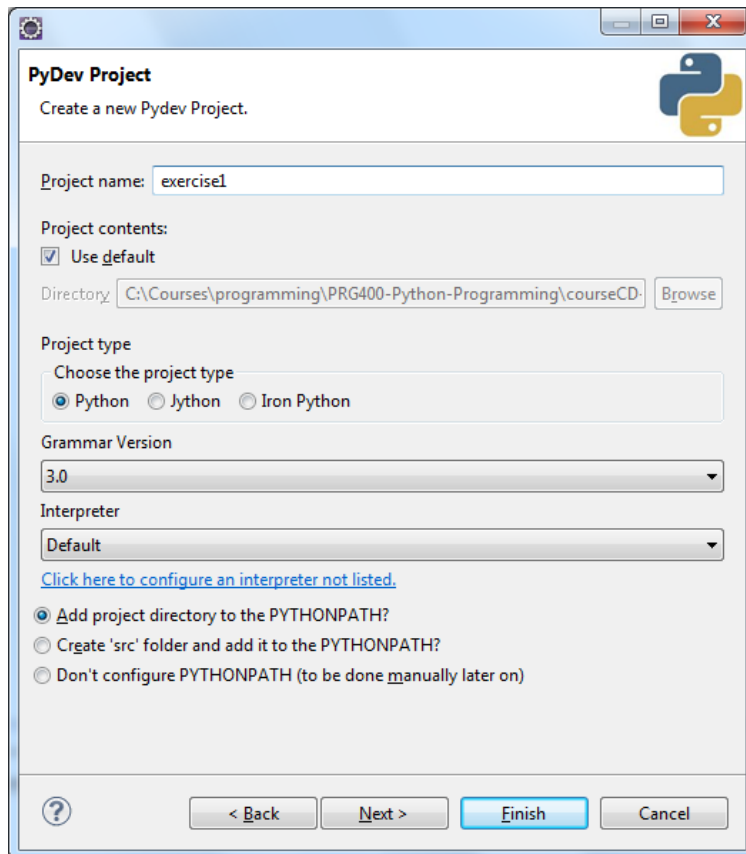
Now configure a Python interpreter by clicking the link. Select new and enter the name and location of the Python interpreter as follows :



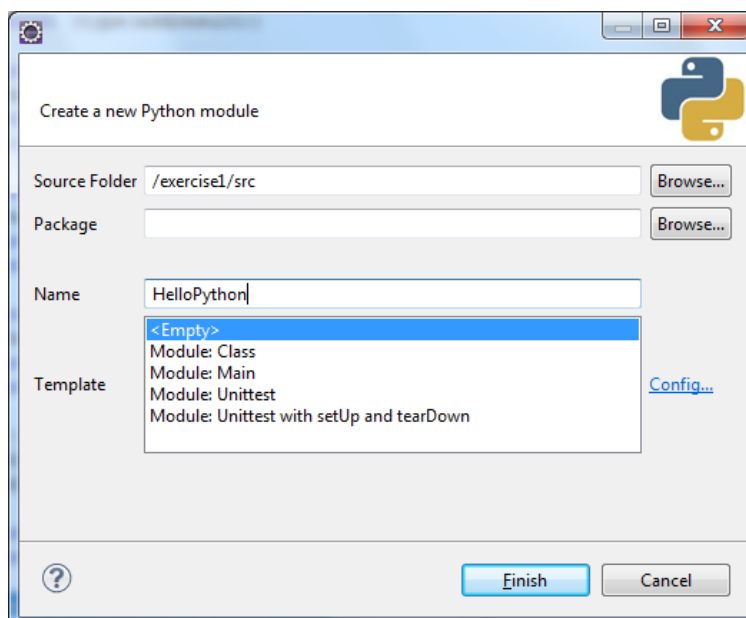
Click OK and keep the default selection of Python folders on the PYTHONPATH. Click OK.



Next choose as project name `exercise1` :



Click Finish. A fresh Python project will open in the Eclipse PyDev perspective. Next select the project node and click New PyDev Module. Give the module the name HelloPython and click Finish.



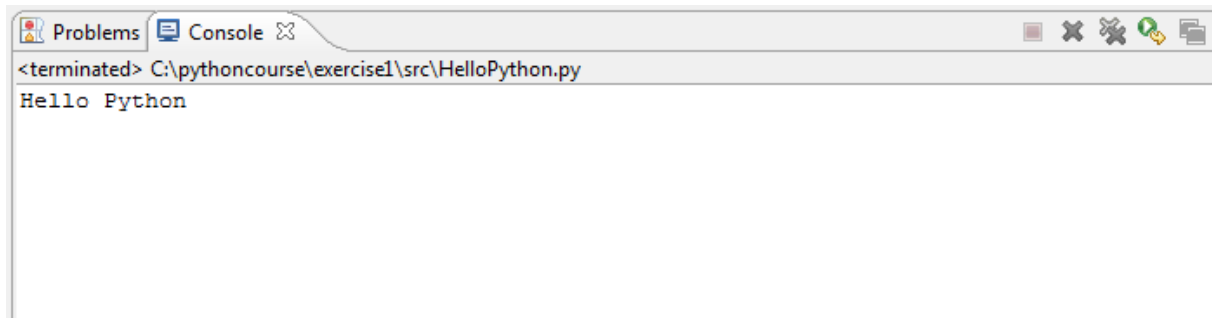
A new source code file `HelloPython.py` will be created in the project. Add the code :

```
print ('Hello Python')
```

to the Python module.

Step 6 : Run a Python Project

Select HelloPython.py and click Run As->Python Run. As a result the following should be displayed in the Eclipse Console :



The screenshot shows the Eclipse IDE's Console window. The title bar includes 'Problems' and 'Console' tabs. The console output displays the command prompt '<terminated> C:\pythoncourse\exercise1\src\HelloPython.py' followed by the output 'Hello Python' on the next line. The console window has standard window controls (minimize, maximize, close) and a refresh icon on the right side of the title bar.

```
<terminated> C:\pythoncourse\exercise1\src\HelloPython.py  
Hello Python
```

Congratulations you have made your first Python program in Eclipse.

Exercise 1 : Python Intro

Open the Python IDLE IDE from the Start menu. What is the version number?

Type `help(str)` :

```
>>> help(str)
```

This command is a simple way to get the documentation for the builtin or standard library class `str`. Alternatively and probably easier you can also use the online HTML documentation or the documentation in WinHelp format that can be found on the course CD. The ">>>" in a listing indicates that the code is being executed in the interactive interpreter. The `C:\` character sequence means executing from the command prompt.

The help on the class `str` in module `built-ins` looks as follows :

```
class str(object)
|   str(string[, encoding[, errors]]) -> str
|
|   Create a new string object from the given encoded string.
|   encoding defaults to the current default string encoding.
|   errors can be 'strict', 'replace' or 'ignore' and defaults to 'strict'.
|
|   Methods defined here:
|
|   __add__(...)
|       x.__add__(y) => x+y
|
```

Note the use of methods with names with two underscores at the beginning and end of the name. These are methods that you will generally never call directly. The `__add__()` method gets executed when you concatenate two strings with the `+` operator.

Now try typing the following commands to see the output. Note that you don't assign a result, you get that result in the interpreter.

```
>>> 'hello world'
'hello world'
>>> _ + '!'
'hello world!'
>>> hw = _
>>> hw
'hello world!'
```

In the interactive interpreter, you can use the special variable `"_"` to refer to the result of the last statement. Handy in this mode, but meaningless in scripts.

Now enter the following into a new file `"hello.py"` in your text editor of choice.

```
print ('hello world!')
```

Save and exit, then execute the script as shown below.

```
C:\ python hello.py
```

Output: `hello world`

You are now invoking the Python interpreter with the Python script `hello.py`.

Exercise 2 : Variables and Data Types

Step 1

Create an Eclipse Project named `exercise2`. Next create a Python module `calculator2.py` in this project and create two variables `nr1` and `nr2`. Assign integer values to them as follows :

```
nr1 = 16
nr2 = 23
```

Use your own values if you like.

Print the values, the type and the memory location of the variables with :

```
print (nr1)
print (nr2)

print (type(nr1))
print (type(nr2))

print (id(nr1))
print (id(nr2))
```

You should see that the type of the variables is `<class 'int'>`. The `id` function gives some numeric value which actually indicates the memory address in a default Python implementation.

Now change the values of the variable by assigning them to a new value and print the memory location of the variables again with the `id` function :

```
nr1 = 16
nr2 = 23
print (id(nr1))
print (id(nr2))
```

Notice that the `id` of the variables has changed just by assigning them. This illustrates the fact that integer variables are read-only. They cannot change once they are created. When you change an integer variable you actually create a new one.

Step 2

Next perform arithmetic operations on the variables by adding, subtracting, multiplying and dividing them. Store the result of these operations in other variables. An example for the addition operation is :

```
addresult = nr1 + nr2
```

Do the same for the other operations.

Display the results of the operations with an explanatory message. An example for the display of the result of the addition is :

```
# result of adding two numbers
```

```
print ("The result of the addition of : " + str(nr1) + " and " + str(nr2) +  
" is : ")  
print (addresult)
```

Do the same for the results of other operations. Notice the conversion function `str` that turns an integer into a string, so that it can be added to other strings.

As an alternative you can also use the string format operator `%` for printing values in string. For the explanatory string for addition this looks as follows :

```
print ("The result of the addition of : %d and %d is : " %(nr1, nr2))
```

Notice that the `str` conversion function is no longer needed. Now we need the string format operator `%` however.

A third alternative is to use the new style of formatting in Python 3 with the `format()` function. Figure out using the course manual or documentation how to use this style of string formatting.

Also display the types of the results. The first three are of `<class 'int'>`. Notice that the type of the division is `<class 'float'>`

Step 3

Now use the `input()` function to ask for the input of two numbers from the user as follows :

```
s1 = input("give me number 1 : ")  
s2 = input("give me number 2 : ")
```

The `input()` function returns a string. Find out about the type of the return values by printing their type as follows :

```
print (type(s1))  
print (type(s2))
```

Notice that the type is : `<class 'str'>`

Now perform addition with the `+` operator on the user input. With user input 1 and 3 the result will be 13, because 1 and 3 are seen as strings. The strings are concatenated.

```
addresult = s1 + s2      # s1=1, s2=3  
print (addresult)       # prints 13
```

Next convert the string to integers with the `int()` conversion function as follows :

```
addresult = int(s1) + int(s2)
```

With the same input the result will now be 4.

When the given input cannot be converted to an `int`, a run time error occurs :

```
ValueError: invalid literal for int() with base 10: 'w'
```

Exercise 3 : Data Structures

Step 1

Create an Eclipse Project named `exercise3` and a Python module `calculator3.py`. Next create an empty list variable and ask the user for the input of two float numbers. Add these numbers, store the result in a variable and append the numbers and the result of their addition to the list.

Repeat this operation until the user quits the program by pressing the red button in Eclipse. Your code could look like the following :

```
reslist = []
while True :
    f1 = float(input("Give me number 1 : "))
    f2 = float(input ("Give me number 2 : "))
    addresult = f1 + f2
    reslist.append(f1)
    reslist.append(f2)
    reslist.append(addresult)
```

Also print the content of the list, print the length of the list and print a subset of the list as follows :

```
print ("Content of list %s : " %(reslist))
print ("Length of list %s : " %len(reslist))
print ("Part of list %s : " %reslist[0:3])
```

Finally remove the first element of the list and print the list again :

```
print (reslist[0])
del reslist[0]
print ("Content of list after deletion %s : " %(reslist))
```

Step 2

Now create a multidimensional list in which each row contains a calculation with as elements the type of operation, the two float numbers and the result of their addition. Also calculate the results of the subtraction, multiplication and division and store these as rows in the multidimensional list. Start by allocating memory for the multidimensional list as follows :

```
multireslist = [[0,0,0,0], [0,0,0,0], [0,0,0,0], [0,0,0,0]]
```

Next ask user input with :

```
while True :
    f1 = float(input("Give me number 1 : "))
    f2 = float(input ("Give me number 2 : "))
```

Then store the results of the operations `add`, `subtract`, `multiply` and `divide` in separate lists and add these list at appropriate places in the multidimensional array. For the `add` operation your code could look as follows :

```
addresult = f1 + f2
addlist = ["Add", f1, f2, addresult]
```

```
multireslist[0] = addlist
```

Write similar code for the subtract, multiply and divide operations.

Print the multidimensional array after storing the results of the different calculations with :

```
print ("Content of list %s : " %(multireslist))
```

Your result should look as follows :

```
Content of list : [['Add', 2.0, 2.0, 4.0], ['Subtract', 2.0, 2.0, 0.0],
['Multiply', 2.0, 2.0, 4.0], ['Divide', 2.0, 2.0, 1.0]]
```

Finally try to change some element in one of the sublists and notice that you are able to make the change e.g. after :

```
multireslist[1][1] = None
```

the contents of the multidimensional list looks as follows :

```
Content of list : [['Add', 2.0, 2.0, 4.0], ['Subtract', None, 2.0, 0.0],
['Multiply', 2.0, 2.0, 4.0], ['Divide', 2.0, 2.0, 1.0]]
```

Finally change one of the sublists to a tuple e.g. :

```
addlist = ("Add", f1, f2, addresult)
```

Notice that when you try to change elements of this tuple you get the following error :

```
TypeError: 'tuple' object does not support item assignment
```

Step 3

Now create an empty dictionary. Ask the user for the input of two numbers like before. Perform add, subtract, multiply and divide operations on the numbers. Store the numbers and the results of the calculation in a small list for each calculation type. Use the following code :

```
resdict = {}
while True :
    f1 = float(input("Give me number 1 : "))
    f2 = float(input ("Give me number 2 : "))

    addresult = f1 + f2
    addlist = [f1, f2, addresult]
```

Now store the list in the dictionary under the key "Add" as follows :

```
resdict["Add"] = addlist
```

Use similar code for the other calculation operations. Print the dictionary after the input of two numbers. You should get the following or similar results :

```
Content of dictionary : {'Multiply': [3.0, 3.0, 9.0], 'Add': [3.0, 3.0,
6.0], 'Subtract': [3.0, 3.0, 0.0], 'Divide': [3.0, 3.0, 1.0]}
```

Try to read and extend the value at dictionary key. The reading will succeed if the dictionary key was created before :

```
resdict["Divide"] = resdict["Divide"] + [1,0,"Infinite"]
```

But you will get an error if the key did not existed before

Next make sure that the keys "Add", "Subtract", "Multiply" and "Divide" exist before starting to ask input of numbers from the user, by initializing the dictionary as follows :

```
resdict = {"Add" : [], "Subtract" : [], "Multiply" : [], "Divide" : []}
```

Now store multiple results of add calculations under the "Add" key as follows :

```
resdict["Add"] = resdict["Add"] + addlist
```

Do the same for the other calculation types. After the input of two sets of numbers you will get the following result when the dictionary is printed :

```
Content of dictionary : {'Multiply': [1.0, 1.0, 1.0, 2.0, 2.0, 4.0], 'Add': [1.0, 1.0, 2.0, 2.0, 2.0, 4.0], 'Subtract': [1.0, 1.0, 0.0, 2.0, 2.0, 0.0], 'Divide': [1.0, 1.0, 1.0, 2.0, 2.0, 1.0]}
```

Step 4

Ask the user to input a list of integers as follows :

```
while True :
    s = input("Give me a set of numbers separated by space characters : ")
```

Now split the input string into a list with :

```
slist = s.split(' ')
print (slist)
```

Also sort and reverse the list with the sort and reverse functions :

```
slist.sort()
print (slist)
slist.reverse()
print (slist)
```

Finally convert the list of strings to a list of integers with :

```
intlist = []
for x in slist :
    x = int(x)
    intlist.append(x)
print (intlist)
```

Feel free to experiment with other string, list and dictionary functions. Use the documentation.

Exercise 4 : Control Flow

Step 1

Create an Eclipse Project named `exercise4` and a Python module `calculator4.py`. Copy the contents of the step 3 of the previous exercise to `calculator4.py`.

Now ask the user, beside the numbers for the calculations, also which type of calculation the user wants to perform. Use `if`, `else` and `elif` statements to determine which branch of the code must be entered. If the user enters "Quit" the application should stop. Your code could look like this for the part that checks whether the application should continue or not :

```
resdict = {"Add" : [], "Subtract" : [], "Multiply" : [] , "Divide" : []}
while True :
    calcAction = input ("What type of calculation do you want to do :")
    if (calcAction == "Quit") :
        break
```

For the part that checks the type of calculation to perform your code could look as follows :

```
else :
    f1 = float(input("Give me number 1 : "))
    f2 = float(input ("Give me number 2 : "))
    if (calcAction == "Add") :
        addresult = f1 + f2
        addlist = [f1, f2, addresult]
        resdict["Add"] = resdict["Add"] + addlist
    elif (calcAction == "Subtract") :
```

Step 2

Now add an option to calculate the sum and average of a list of numbers. Copy part the code from step 4 of exercise3 to convert a list of strings to a list of integers :

```
slist = s.split(' ')
intlist=[]

for x in slist :
    x = int(x)
    intlist.append(x)
```

Based on the input of the desired calculation type, you will have to make a distinction between the input of a list of numbers or only two numbers. Use code like the following :

```
elif ((calcAction == "Sum") or (calcAction == "Average")) :
    s = input("Give a set of numbers separated by space characters : ")
and

elif ((calcAction == "Add") or (calcAction == "Subtract")
      or (calcAction == "Multiply") or (calcAction == "Divide")) :
```

```
    f1 = float(input("Give me number 1 : "))
    f2 = float(input ("Give me number 2 : "))
```

Now calculate the sum or the average of the list of numbers in a for loop. Don't forget to declare variables for the sum and the average first. Do not use sum as a variable name since this is a reserved word. Use summation instead. Finally print the result of the calculation. For the calculation of the sum the code could look as follows :

```
summation = 0
average = 0
if (calcAction == "Sum") :
    for x in slist :
        x = int(x)
        summation = summation + x
        intlist.append(x)
    print ("Sum of list %s is %d" %(str(intlist), summation))
```

Step 3

Replace the while True statement at the beginning of the code by a test on the input of the calculation type. Stop the while loop when the input is "Quit". Use code like the following :

```
calcAction = "Continue"
while (calcAction != "Quit") :
    calcAction = input ("What type of calculation do you want to do :"):
```

Also add an else statement to the while loop that gets executed when the while loop stops. Use code like the following :

```
else :
    print ("Thank you for user our software")
```


Exercise 5 : Functions

Step 1

Create an Eclipse Project named `exercise5` and a Python module `calculator5.py`. Copy the contents of the step 3 of the previous exercise to `calculator5.py`.

Write functions to add, subtract, multiply and divide two numbers and replace the code that does the calculations inline by a call to these functions. The code for the `add` function could look as follows :

```
def add (i, j):  
    z = i + j  
    return z
```

and calling this function is done with :

```
addresult = add(f1, f2)
```

Step 2

Write a function `convertointlist` that converts a list of input character to a list of integers. Your code could look as follows :

```
def convertointlist(slist):  
    ilist = []  
    for x in slist :  
        x = int(x)  
        ilist.append(x)  
    return ilist
```

Call this function with as parameter the input list given by the user for the sum and average calculations :

```
s = input("Give a set of numbers separated by space characters : ")  
slist = s.split(' ')  
intlist = convertointlist(slist)
```

Also write a function `summate` that takes a list of integers as an argument, and returns the sum of the integers in that list. And write a function `average` that takes a list of integers and returns the average of these integers. For the `summate` function your code could look as follows :

```
def summate(ilist):  
    s = 0  
    for x in ilist :  
        s = s + x  
    return s
```

Replace the code that does the summation code and averaging code inline by a call to these functions.

Step 3

Add another calculation option to the calculator called "Faculty". When the user enters this, one number should be entered and the faculty of this number should be calculated. Hereby the faculty of n is defined as $n * (n-1) * (n-2) * (n-3) * \dots * 1$

Create a recursive function called `faculty` to perform this calculation and call this function.

Exercise 6 : Modules

Step 1

Create an Eclipse Project named `exercise6` and a Python module `calculator6.py`. Copy the contents of the `calculator5.py` from the previous exercise to `calculator6.py`.

Next create a Python module `calculations.py`. Remove the function definitions from `calculator6.py` and put them into `calculations.py`. As a result all function calls in `calculator6.py` will give errors because they are calling undefined functions.

Resolve the errors by importing the `calculations` module with an `import` statement in the first line of `calculator6.py`:

```
import calculations
```

This is not enough. You will also need to prefix all function names with the module name where they come from :

```
addresult = calculations.add (f1, f2)
```

Step 2

It is inconvenient to put the module name as prefix before the function name. Therefore there is an alternative syntax called `from <module> import <name>`. Now change to this alternative and name all the function names that must be imported in the `from import` statement :

```
from calculations import faculty, convertointlist, summate, average, add, subtract, multiply, divide
```

As a result the prefixes before the method calls can be removed in `calculator6.py`.

Step 3

Finally make a `calcu` package. Put the `calculations.py` module in this package, edit the `__init__.py` file and import the functions from the package in `calculator6.py`.

Exercise 7 : Classes and Objects

Step 1 : Employee class

Make an `Employee` class representing some employee working in a company in the python module `employee.py`. The `Employee` class should have two attributes representing the name and the ID of an employee.

Add a constructor to the `Employee` class which sets the attributes when they are passed. Make a simple program which declares an `Employee` object and display the values of the attributes of the `Employee` object by accessing the attributes directly. Use code like the following :

```
class Employee(object) :
    def __init__(self, name):
        print ("Calling Employee constructor")
        self.name = name
```

Next make the attributes private in the `Employee` class and add public set and get member functions in the `Employee` class to be able to access these attributes. Change the values of the attributes through the accessor functions and display their changed values. Use code like the following :

```
def set_name(self, value):
    self.__name = value
def del_name(self):
    del self.__name
```

Notice that private data cannot be accessed directly from outside of the class. This demonstrates the object oriented principle of encapsulation. In Python there is a workaround so you can still access private data. Use this workaround also for demonstration purpose.

Also add a destructor to the class and keep a class variable which counts how many instances of the class are present. Use code like the following :

```
def __del__(self):
    class_name = self.__class__.__name__
    print (class_name, "destroyed")
```

Write print statements when the constructor or destructors and the getters and setters are called. Also print an instance variable directly. What is the result?

Next make a property in the class with the following code :

```
name = property(get_name, set_name, del_name, "name's docstring")
```

Access the `Employee` object through the property as follows :

```
e1.name = "Some Name"
print (e1.name)
```

Also write statements to show the class and object data :

```
print ("Employee.__doc__:", Employee.__doc__)
print ("Employee.__name__:", Employee.__name__)
```

```
print ("Employee.__module__:", Employee.__module__)
print ("Employee.__bases__:", Employee.__bases__)
print ("Employee.__dict__:", Employee.__dict__)
```

Put the Employee class in a script in the employees package and add code to be able to export the Employee class from this package.

Give the Employee class a `calcSal()` method as follows :

```
def calcSal(self):
    return 0
```

Optionally you can also add a `hiringdate` attribute to the Employee class.

Add the following code to test the Employee class in stand alone mode :

```
if __name__ == '__main__':
    main()
```

Step 2 : Company Script

Make a Company script above the employees package.

Add the necessary import statements to be able to find the Employee class in the employee script in the employees package. If you keep the `__init__.py` file in the package empty your code could look as follows :

```
import employees.employee

e1 = employees.employee.Employee("Albert Einstein")
```

Create a number of employees and put them in a list, called for instance `employeeList`. Next iterate through the list and print the name, id and salary of the employees in the list.

If time permits :

Alternatively you could create a `Company` class with functions to add or to remove employees from the `employeeList` which displays a list of employees including their name, ID. In that case make a simple script which declares a `Company` object, add several employees to the company and display a list of employees.

The relationship between the `Company` class and the `Employee` class is an association relation of the normal type. We can say every `Company` employs `Employees` or every `Employee` works for a `Company`.

Step 3 : Inheritance

Derive a `WageEmployee` class from the `Employee` class. The `WageEmployee` class inherits the attributes and public methods of the `Employee` class. Define two additional attributes in `WageEmployee` `wage`, representing the hourly wage, and `hours`, representing the hours worked. Both these attributes should be private. Also implement `get`, `set` and `del` methods for these attributes. Your code could look as follows :

```
def get_wage(self):
    return self.__wage
```

```

def get_hours(self):
    return self.__hours
def set_wage(self, value):
    self.__wage = value
def set_hours(self, value):
    self.__hours = value
def del_wage(self):
    del self.__wage
def del_hours(self):
    del self.__hours

```

Put the WageEmployee in the same package as the Employee.

Further define a calcSal method in WageEmployee which returns the salary of a WageEmployee by multiplying hours and wage. Your code could look as follows :

```

def calcSal(self):
    return self.__wage * self.__hours

```

The relationship between a WageEmployee and an Employee is a generalization / specialization, also called an inheritance relation. A WageEmployee is a special type of Employee.

Add a constructor to the WageEmployee class which accepts type the name and id attributes. Use this constructor also to set the hours and wage attributes to default values. Notice that in order to fill the name and id attribute an appropriate Employee constructor must be called. It is not possible to access the attributes directly. This constructor should already be present in the Employee class. Call this constructor from the WageEmployee constructor. Use code like the following :

```

class WageEmployee(Employee) :
    def __init__(self, name, wage, hours):
        super(WageEmployee, self).__init__(name)
        print ("Calling WageEmployee constructor")
        self.__wage = wage
        self.__hours = hours

```

Also make wage and hours properties as follows :

```

wage = property(get_wage, set_wage, del_wage, "wage's docstring")
hours = property(get_hours, set_hours, del_hours, "hours's docstring")

```

Next declare an instance of a WageEmployee in the simple program which also declares the Company object and call its methods. Or do this in the script if you did not create the Company class. Try to use the private name and id attributes from the WageEmployee class. You will not succeed. Use the accessor methods instead. Your code could look like the following :

```
w1 = employees.wageemployee.WageEmployee("Werner Heisenberg", 10, 100)
print (w1.name)
print (w1.wage)
print (w1.hours)
```

Add a `ManagerEmployee` to the `Employee` hierarchy which is derived `Employee` and whose salary is a fixed value per month. Add a private `monthlySalary` attribute of the integer type to the `ManagerEmployee` class. Its `calcSal` method should return the value of that attribute. It should be possible to give the `ManagerEmployee` a salary in its constructor. Add a constructor and add accessor functions for the `monthlySalary` attribute. Do something like the following :

```
class ManagerEmployee(Employee) :
    def __init__(self, name, weeksal):
        super(ManagerEmployee, self).__init__(name)
        print ("Calling ManagerEmployee constructor")
        self.__weeksal = weeksal
```

Put the `ManagerEmployee` in the same package as the `Employee`.

Add a `ProgrammerEmployee` to the `Employee` hierarchy which is derived from `WageEmployee` and whose salary consists of the `WageEmployee` salary + a percentage of the lines produced, minus a percentage of the bugs introduced. Add attributes `lines` and `bugs`

Add a constructor and add accessor functions for the `lines` and `bugs` attributes.

Conceive an appropriate implementation of its `calcSal` method

Put the `ProgrammerEmployee` in the same package as the `Employee`.

Also add destructors for all these classes.

Step 4 : Polymorphism

Create instances of each of the derived `Employee` classes in the script which declares the `Company` object or in the script without the `Company` object.

Give each employee in the hierarchy, a name, id, and a salary and display these. Next add all the employees to the `employeeList` in the `Company` object. Make a function `payroll` in the `Company` object which iterates through the list of employees, calculates each of their salaries, adds them and finally returns the total payroll of the `Company`.

Calling the `calcsal` method on the employees in the list, while executing different versions of these `calcsal` method depending on the concrete derived `Employee` in the list, is a demonstration of the object oriented principle of polymorphism.

In Python it is sufficient for all classes to have the same method to enable polymorphism. In Java or C++ the `calcSal()` method must be present in the base class.

If time permits :

Add code that limits the permitted values for the ID attribute to values between 1 and 1000. If someone tries to set the value higher or lower than these limits an error message should be displayed or an exception should be raised and the ID value should be set to the nearest permitted value. The check can be done in the `set` method and the constructor.

Exercise 8 : Exception Handling

Step 1

Create an Eclipse Project named `exercise8` and a Python module `calculator8.py`. Copy the contents of the `calculator6.py` from the previous exercise to `calculator8.py`. Also copy the `calculations` module.

Write `try: except:` statements for the following cases. In all these cases the current program crashes. With exception handling we can ask for new input.

- 1) Division by zero.
- 2) Wrong input for Add, Subtract, Multiply and Divide operations.
- 3) Wrong input for the number for the factorial
- 4) Wrong input for the list of integers.

Step 2

Define an exception class. Then write a `try:except:` statement in which you throw and catch that specific exception.

Exercise 9 : Python IO

Step 1

Create an Eclipse Project named `exercise9` and a Python module `calculator9.py`. Copy the contents of the `calculator8.py` from the previous exercise to `calculator9.py`. Also copy the calculations module. You will use the solution code of the previous exercise as a starting point.

Open a text file for writing just before entering the while loop that asks the user to choose a calculation type. Use the following code :

```
fw = open('calculationdata.txt', mode='w', encoding='UTF-8')
```

Next add the details of a calculation to a line in this file when each calculation is performed.

For the Faculty calculation you could use the following code :

```
fw.write("%s,%s,%s\n" % (calcAction, x, faculty(x)))
```

For the Sum and Average calculations you could use the following code :

```
fw.write("%s,%s,%s\n" % (calcAction, str(intlist), summate(intlist)))
```

And for the Add, Subtract, Divide and Multiply calculations you could use the following code just after the input of the numbers :

```
f1 = float(input("Give me number 1 : "))
f2 = float(input ("Give me number 2 : "))
fw.write("%s,%s,%s," % (calcAction, str(f1), str(f2)))
```

And the following code just after each calculation :

```
fw.write("%s\n" % str(add(f1, f2)))
```

Finally close the file when you give the "Quit" command otherwise nothing is written.

```
fw.close()
```

Now enter some calculations to test your code and observe the contents of the file.

Step 2

Next create a new Python module `readcalculations.py`, read the lines of the file and display its contents with the following code :

```
fr = open('calculationdata.txt', mode='r', encoding='UTF-8')
for line in fr.readlines():
    s = line.rstrip('\n')
    slist = s.split(',')
    print (slist)
fr.close()
```

The results could look as follows :

```
['Faculty', '12', '479001600\n']
['Sum', '[1', '1', '3]', '5\n']
['Average', '[4', '5', '6]', '5.0\n']
```


Exercise 10 : Python Database Access

Step 1

Create an Eclipse Project named `exercise10` and a Python module `calculator10.py`. Copy the contents of the `calculator9.py` from the previous exercise to `calculator10.py`. Also copy the calculations module. You will use the solution code of the previous exercise as a starting point and replace the file access code with database access code.

Before entering the `while` loop in which the user is asked to choose the type of calculation he wants to perform use the following code to create a database `calculations.db` and a table `Calculations` :

```
con = sqlite3.connect('calculations.db')

with con:
    cur = con.cursor()
    cur.execute("DROP TABLE IF EXISTS Calculations")
    cur.execute("CREATE TABLE Calculations
                (Id VARCHAR(10), Nr1 INT, Nr2 INT, Result DECIMAL)")
```

Next consider the Faculty calculation type and replace the code where data is written to the file with code that writes data to the database. Write the following code on the same line :

```
with con:
    cur = con.cursor()
    cur.execute("INSERT INTO Calculations ('Id', 'Nr1', 'Nr2', 'Result')
                VALUES(?,?,?,?)", (calcAction, x, 0, faculty(x)))
```

In this exercise we will not consider the Sum and Average calculation types. Just remove the code that writes data to the file. We will however consider the Add, Subtract, Divide and Multiply calculation types. For these calculation types replace the file access code with the following database access code. For instance for the Add calculation type :

```
with con:
    cur = con.cursor()
    cur.execute("INSERT INTO Calculations ('Id', 'Nr1', 'Nr2', 'Result')
                VALUES(?,?,?,?)", (calcAction, f1, f2, add(f1, f2)))
```

Next insert some records in the database, observe the existence of the `calculations.db` file in the directory of the script and use the SQLiteManager plugin of FireFox to connect to SQLite and examine the contents of the database. The table contents could look as follows :

Run SQL

Actions ▾

Last Error: not an error

Id	Nr1	Nr2	Result
Faculty	5	0	120
Add	1	1	2
Subtract	2	3	-1
Multiply	5	6	30
Divide	100	10	10

Exercise 11 : Python and XML

Step 1 : minidom

Create a Python script that creates an XML structure with the xml minidom library looking as follows :

```
<?xml version="1.0" ?>
<employees>
  <employee type="emp">
    <name>
      Albert Einstein
    </name>
    <id>
      11111
    </id>
    <ssn>
      123456789
    </ssn>
    <age>
      55
    </age>
    <hiringDate>
      11-12-1953
    </hiringDate>
  </employee>
  <employee type="wageemp">
    <wage>
      10
    </wage>
    <hours>
      10
    </hours>
  </employee>
  <employee type="manageremp">
    <weeksal>
      10
    </weeksal>
  </employee>
  <employee type="programmeremp">
    <lines>
      10
    </lines>
    <bugs>
      10
    </bugs>
  </employee>
</employees>
```

Step 2 : Sax

Use the xml.sax library to parse and display the elements of an XML file.

Exercise 12 : Python Libraries

You will find documentation on the use of the Python libraries in the root directory of the courseCD.

Step 1

Make a program that waits for user input and that reads a licence plate for a car with the input function. A licence plate of a car should have the following format : 2 uppercase character, a dash, 2 numbers , a dash and two uppercase characters again like in : AB-67-NK. Verify that the license plate is correct with a regular expression.