Introduction to Python

Session 10

Introduction to Object Oriented Programming (OOP)

Exceptions

Exceptions


```
x = int(input("Please enter a number: "))
```

If the input is not an integer, the conversion cannot be done.
An exception is produced.

upf.

Exceptions


```
while True:

   try:
      x = int(input("Please enter a number: "))
      break
   except Exception:
      print("Oops! That was not a valid number.
            Try again…")
```

In this case, the exception is **handled**.

upf.

Exceptions


```
try:
    ...
    ...
except Exception:
    ...
    ...
```

Block of statements executed.

Block of statements executed if an error exception is produced.
If there is no exception, this block is skipped

upf.

Exceptions


```
while True:

   try:
      x = int(input("Please enter a number: "))
      break
   except ValueError:
      print("Oops! That was not a valid number.
            Try again…")
```

In this case, the **exception is handled.**

upf.

Exceptions

Not capturing the correct exception: if a ValueError exception is raised the script is only capturing a TypeError exception

```
while True:

    try:
        x = int(input("Please enter a number: "))
        break
    except TypeError:
        print("Oops! That was not a valid number.
              Try again…")
```

In this case, the **exception is NOT handled.**

Exceptions


```
try:
    ...
    ...
except (ValueError, TypeError, NameError):
    pass
```

Multiple except clauses captured

upf.

Exceptions


```
try:
        f = open("myfile.txt","r")
        s = f.readline()
        i = int(s.strip())

except IOError as e:
        print("I/O error (%s): %s" %(e.errno, e.strerror))

except ValueError:
        print("Could not convert data to an integer.")
except:
        print("Unexpected error")
        raise
```

In this example, **different exceptions are handled differently**

Exceptions


```
try:
        f = open("myfile.txt","r")
        s = f.readline()
        i = int(s.strip())
except IOError as e:
        print("I/O error (%s): %s" %(e.errno, e.strerror))
except ValueError:
        print("Could not convert data to an integer.")
except:
        print("Unexpected error" )
        raise
```

Capture the exception object and assign it to a name (e)

In this example, **different exceptions are handled differently**

Exceptions


```
try:
        f = open("myfile.txt","r")
        s = f.readline()
        i = int(s.strip())
except IOError as e:
        print("I/O error (%s): %s" %(e.errno, e.strerror))
except ValueError:
        print("Could not convert data to an integer.")
except:
        print("Unexpected error")
        raise
```

Access attributes of the exception instance named e
Specific attributes of IOError!

In this example, **different exceptions are handled differently**

upf.

Exceptions


```
def this_fails():
    x = 1/0

try:
    this_fails()
except ZeroDivisionError as detail:
    print("Handling run-time error: %s" %detail)
```

Capture the exception object ZeroDivisonError and assign it to the name detail

In this example, the exception is produced in the function `this_fails()`

As the exception is not handled inside the function, it is transmitted until some code handles it.

upf.

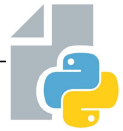
Exceptions


```
def function3():
        print("I am executing function 3")
        return 1/0

def function2():
        print("I am executing function 2")
        return function3()

def function1():
        print("I am executing function 1")
        return function2()


function1()
```



upf.

Exceptions

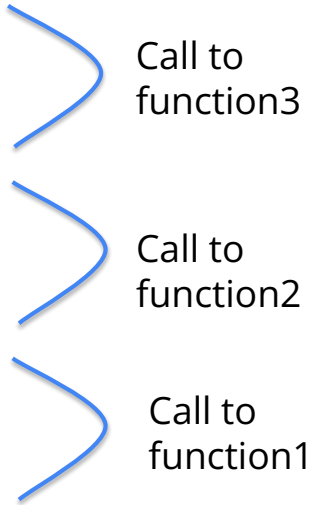

```
import sys

def function3():
        print("I am executing function 3")
        return 1/0

def function2():
        print("I am executing function 2")
        return function3()

def function1():
        print("I am executing function 1")
        return function2()


function1()
```



Exceptions


```
import sys

def function3():
        print("I am executing function 3")
        return 1/0

def function2():
        print("I am executing function 2")
        return function3()

def function1():
        print("I am executing function 1")
        return function2()


function1()
```

Exception!!!

Exception transmitted

Exception transmitted

Exception transmitted

upf.

Exceptions

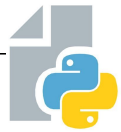

```
I am executing function 1
I am executing function 2
I am executing function 3
Traceback (most recent call last):
  File "sample_exceptions.py", line 16, in <module>
    function1()
  File "sample_exceptions.py", line 13, in function1
    return function2()
  File "sample_exceptions.py", line 9, in function2
    return function3()
  File "sample_exceptions.py", line 5, in function3
    return 1/0
ZeroDivisionError: integer division or modulo by zero
```

In standard error we can see all the traceback of the exception flow!

→ Very useful for **debugging**!

upf.

Exceptions



```
import sys

def function3():
        print("I am executing function 3")
        try:
          return 1/0
        except ZeroDivisionError as e:
          print(“Trying to divide by 0!”)
          return None

def function2():
        print("I am executing function 2")
        return function3()

def function1():
        print("I am executing function 1")
        return function2()

function1()
```

Exception!!!

The exception is handled in function 3 and not transmitted

upf.

Exceptions


```
I am executing function 1
I am executing function 2
I am executing function 3
Trying to divide by 0!
```

The exception is handled in function 3 and not transmitted

upf.

Exceptions

To create an Exception instance:

```
e = ValueError("this is the message of the exception")
print(e)
This is the message of the exception
print(type(e))
ValueError
```

To throw an exception: **raise**

```
e = ValueError("Message of the error")
raise e
```

```
raise NameError("Message of the error")
```

upf.

Exceptions

finally

The finally clause is always executed before leaving the try statement, whether an exception has occurred or not

Useful for closing the files whether an exception was raised or not

```
try:
    fd = open("my_file.txt","r")
    X = 1 / 0

finally:
    fd.close()
    print("Closing opened files")
```

upf.

Exceptions

try
except
else
finally

else clause executes code that must be executed if the try clause does not raise an exception

```
def divide(x, y):
    try:
        result = x/y
    except ZeroDivisionError:
        sys.stderr.write("Dividing by 0. Returning infinite\n")
        result = float("Inf")
    else:
        print("result is %s" %result)
    finally:
        print("executing finally clause")
    return result
```

upf.

Exceptions

else
finally

```
def divide(x, y):
    try:
        result = x/y
    except ZeroDivisionError:
        sys.stderr.write("Dividing by 0. Returning infinite\n")
        result = float("Inf")
    else:
        print("result is %s" %result)
    finally:
        print("executing finally clause")
    return result
```

else clause executes code that must be executed if the try clause does not raise an exception

```
>>> divide(2,1)
result is 2.0
executing finally clause
2.0
```

upf.

Exceptions

else
finally

```
def divide(x, y):
    try:
        result = x/y
    except ZeroDivisionError:
        sys.stderr.write("Dividing by 0. Returning infinite\n")
        result = float("Inf")
    else:
        print("result is %s" %result)
    finally:
        print("executing finally clause")
    return result
```

else clause executes code that must be executed if the try clause does not raise an exception

```
>>> divide(2,0)
Dividing by 0. Returning infinite
executing finally clause
inf
```

upf.

Exceptions


```
def divide(x, y):
    try:
        result = x/y
    except ZeroDivisionError:
        sys.stderr.write("Dividing by 0. Returning infinite\n")
        result = float("Inf")
    else:
        print("result is %s" %result)
    finally:
        print("executing finally clause")
    return result
```

```
>>> divide("2","1")
executing finally clause
Traceback (most recent call last):
  File "sample_exceptions4.py", line 14, in <module>
    divide("2","1")
  File "sample_exceptions4.py", line 3, in divide
    result = x / y
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

upf.

classification of exceptions that are in python!

Exceptions


```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Warning
       +-- DeprecationWarning
       +-- PendingDeprecationWarning
       +-- RuntimeWarning
       +-- SyntaxWarning
       +-- UserWarning
       +-- FutureWarning
       +-- ImportWarning
       +-- UnicodeWarning
       +-- BytesWarning

+-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
      |              +-- UnicodeDecodeError
      |              +-- UnicodeEncodeError
      |              +-- UnicodeTranslateError
```

upf.

Defining our own exceptions


```
class MyError(Exception):
    """My defined exception"""
    pass
```

```
e = MyError()

raise e
```

upf.

Defining our own exceptions


```
class MyError(Exception):
    """My defined exception"""

    def __init__(self, value):
        self.value = value
        super().__init__(f"Incorrect assigned value: {value}")
```

Creates a new data attribute: self.value

def __init__(self, value):: This is the constructor. It takes an input (value) and stores it in the object so you can access it later.

Exercises. Session 10

uID_S10.py

1) Create a new ValueError exception subclass named **`IncorrectSequenceLetter`**

2) The new exception instance should be created with the letter not found in the alphabet and the class name of the sequence. Example:

this letter does not match the aphabet

```
e = IncorrectSequenceLetter("B", class_name)
```

class name is protein, dna or rna??

3) The description string of the exception must be the following:

"The sequence item *B* is not found in the alphabet of class *ProteinSequence*"

4) Sequence class should raise an **`IncorrectSequenceLetter`** exception when a sequence is created using an incorrect letter not found in the alphabet.

protein, dna or rna

5) Modify the `FASTA_iterator` to be able to iterate in any type of Sequence (add a new argument in the generator function to specify the Sequence class to use)

upf.

Exercises. Block 2. Part 4.

letters don't match the alphabet, so when a seq have an incorrect eltter, the code is stoped as well as its execution

6) Modify the `FASTA_iterator` generator function to **skip sequences having incorrect letters** . It must capture specifically the `IncorrectSequenceLetter` exception. When it happens, it should print a message of error in the **standard error** and continue with the next sequence. Do not handle other types of exceptions, only IncorrectSequenceLetter exceptions.

creating our own incorrect seq and modify the FASTA_iterator to deal with that

try except

7) When the script is executed as a **standalone** application (without being imported (i.e. code under __main__ block), the script should read input DNA FASTA file(s) and calculate the **length** and **molecular weight** of their corresponding proteins (i.e. corresponding to ProteinSequence instances obtained **after translation)**. The script must print the output to **standard output** or to a **file**. Output should be sorted by molecular weight, from lowest to greatest.

not imported to another script

```
Python uID_S10.py [IN] [OUT]
```

Exercises. Block 2. Part 4.

7.1) If the script is executed **without arguments**, it looks for all ".fasta" or ".fa" files in the current directory, and process all of them with a single sorted output. Print the results to **standard output**.

7.2) If the script has a **single argument,** it corresponds to the input. If it is a directory, it looks for all ".fasta" or ".fa" files in the given directory, process them and print the results in standard output. If this single argument corresponds to a file (not necessarily ".fasta", or ".fa"), process it and print the results in **standard output**.

7.3) If the script has **two arguments**, the first one corresponds to the input and the second one to the output file.

7.4) The script should print to **standard error** a progression **log**

Exercises. Block 2. Part 4.

Input FASTA format:

```
>Identifier
ATCTACTGAACGTTAAGACTACTACACTTTACAT
CTAGCCCCATTGACGATACGTTAAA
>Identifier2
ACAATACGGGTACCAGAT
```

Multiline fasta file

Progression log printed to **standard error**:

```
3 FASTA files found.
input_path/Fasta_file1.fasta finished.
input_path/Fasta_file2.fasta finished.
input_path/another_file.fa finished.
1213 sequences found.
Sorting the sequences…
Sort process finished.
Program finished correctly.
```

How many fasta files are going to be processed.

For each FASTA file, indicate the process has finished.

Indicate when the sorting starts, when it ends and when the program has finished.

upf.

Exercises. Block 2. Part 4.

Output format

```
Identifier1   Length1   molecular_weight1
Identifier2   Length2   molecular_weight2
Identifier3   Length3   molecular_weight3
```

Exercises. Block 2. Part 4.

Hints:

1) Modify the `FASTA_iterator` function in **so it creates an instance of a specific subclass** (Until now, it created only ProteinSequence instances. To be useful for different scenarios it should create instances of a specific Sequence subclass. To achieve this, you can add an additional argument to the FASTA_iteration function to specify the subclass to be created).

2) If the input FASTA files are incorrect, i.e., for example, incorrect sequence monomers, skip the incorrect sequences by managing the exceptions.

upf.