FILES I/O AND EXCEPTIONS

ES 112

Brief Recap

- Functions as First Class Objects
- Functional Programming: map, filter and reduce
- List Comprehensions

Menu for Today!

- Iterators
- File Handling
- Exceptions

ITERATORS

Iterables and Iterators

- Iterators
  - *A class of objects that implement a special method* `__next__`
  - `__next__` *yields the next element until no more elements are left*
- Iterables
  - *Formally, a class of objects that implements a special method* `__iter__` *that returns an iterator*

```
numbers = range(1,5)
numIter = iter(numbers)          #numIter = numbers.__iter__()
print(next(numIter))             #print(numIter.__next__())
```

BuiltIn Iterators

- the expression `for i in iterator` implicitly invokes `__next__` every time it is executed
- We have seen this behavior in our discussion of loops using
  - *Lists*
  - *Tuples*
  - *Dictionaries*
  - *Range*
  - *String*

Using Iterators

- `for` loops
- `map` and `filter` objects
- List comprehension
- Three iterators from a dictionary
  - `for` *iterates over keys*
  - `dictionary.values()` *returns an iterator over values*
  - `dictionary.items()` *returns an iterator over keys and values*
- Create an iterable data structure using `list`, `tuple` etc
  - *Remember* `map` *and* `filter` *objects? Applying* `list` *to these objects allowed us to create a list*

Generator Objects

- Using comprehension syntax, we can create `generator` objects
  - `generator` *objects are* `iterable`

```
genObject = (expression for item in iterable)
for i in genObject
    print(i)
```

FILE HANDLING

Handling Files

- Taking input from `stdin`, and writing output to `stdout` is tedious.
- We would like to read input from, and write output to files
- To read from a file, we first need to open a file and bind it to a file handler

```
fh = open('myFile','w')
```

- 'w' specifies the mode we have opened the file in : here we intend to write to this file

```
fh.write("This is some spam text")
```

- Always remember to close the file before you exit the program

```
fh.close()
```

An Example

```
profsFile = open(‘profs’, ‘r’)
prof1 = profsFile.readline()
prof2 = profsFile.readline()
profsFile.close()
courseFile = open(‘es112’,’w’)
courseFile.write(f’ES112 lectures are taught by {prof1}’)
courseFile.write(f’ES112 labs are taught by {prof2}’)
courseFile.close()
```

Some Useful File Functions and Modes

- `fh.read()` : read a string from file `fh`
- `fh.readline()` : read the next line from file `fh`
- `fh.readlines()` : read all the remaining lines from file `fh` and return a list of strings
- `fh.write(s)` : write string s to the end of file `fh`
- `fh.writelines(S)` : given a list S of strings, write each element of S as a separate line to file `fh`
- Common file opening modes:
  - *`r` : open an existing file for reading*
  - *w : open a file for writing. If the file exists, it will get overwritten*
  - *`a` : open an existing file for appending. New content is written after the existing content*

EXCEPTIONS

Catching Runtime errors

- Recall our first conditional (a long time ago)

```
if (divisor != 0):
     print(dividend, " / ", divisor, " = ", dividend / divisor)
else:
    print("Cannot divide by zero")
```

- Here, we were trying to deal with a run-time error
  - *We first check for error conditions (`divisor == 0`)*
  - *If the error condition if false, we execute the intended code*
  - *If the error condition is true, we printed an error message*

Problems with this Approach

- We will need to write this error checking code every time we attempt any operation
  - *The error could happen anywhere, including inside a (built-in) function*
  - *The error handling on the other hand may be far removed from where the error happens – after the return from the function call perhaps*
- We may want to do something more than simply printing an error message on encountering an error
  - *We may want to keep track of all the values that resulted in errors*
- The Python mechanism for marking something as an error is to `raise` an `exception`
- The Python mechanism for handling an exception is using `try`

Handling Errors during a Function Call

- Remember this function from earlier:

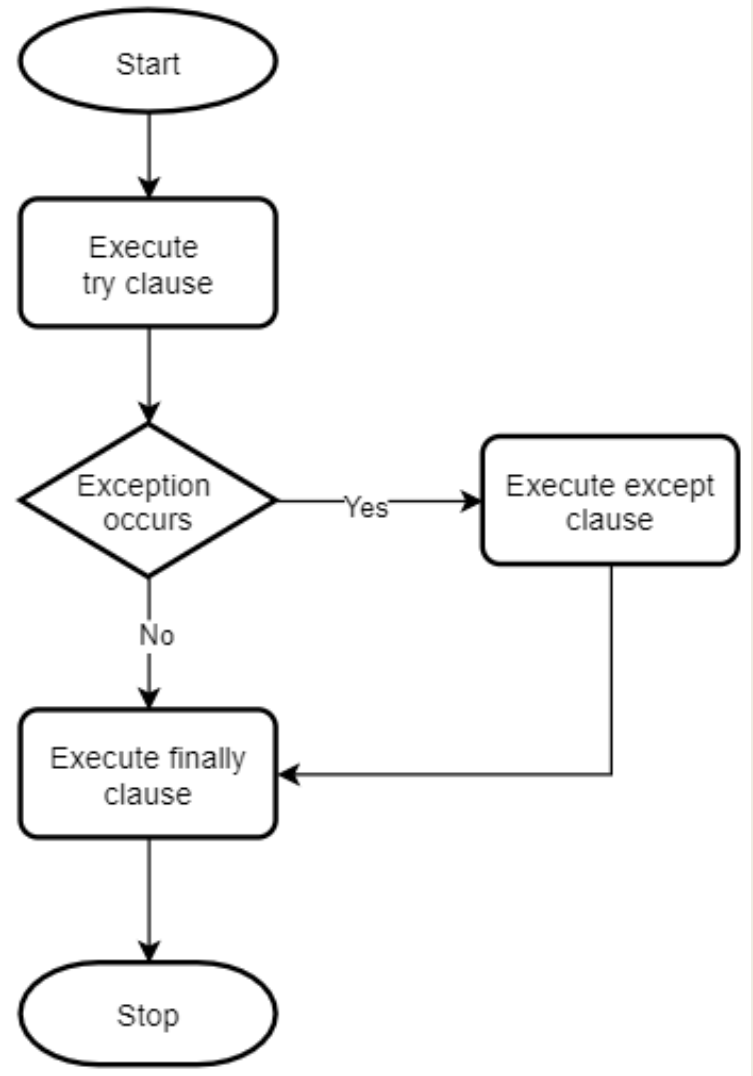
```
def promptForIntegerInput():
    data = int(input(“Give me an integer between 0 and 100”))
    if (data < 0 or data > 100):
        print(f’{data} is out of range’)
    else:
        return data
```

- What should the caller do if the data is out of range?

Raising an Exception

```
def promptForIntegerInput():
    data = int(input(“Give me an integer between 0 and 100”))
    if (data < 0 or data > 100):
        raise Exception(‘data is out of range’)
    else:
        return data
try:
    myInput = promptForIntegerInput()
except:
    print(‘Input is out of range’)
```

Raising an Exception: A Flowchart



Division by Zero

```
try:
    print(dividend, " / ", divisor," = ", dividend / divisor)
except:
    print("Cannot divide by zero")
```

- Attempting division by zero raises a (built in) exception called `ZeroDivisionError`
- This exception is handled by the code after the `except` clause

Maintaining a List of Values that Gave Errors

```
stateCapitals = {'KA': 'BLR', 'MH': 'BOM', 'AP': 'HYD',
'MYSORE': 'MYSORE' }
queryList = []
def myFunc(myKey):
    try:
        return(stateCapitals[myKey])
    except:
        queryList.append(myKey)
        return(None)
myFunc('DEL')
```

Better Control on Error Handling

- A bare try clause will ”handle” all errors; this may not be what we want
  - *Better control on exception: use named exceptions*
  - *Builtin Exceptions: AssertionError, AttributeError, FloatingPointError, MemoryError, IndexError, NotImplementedError, NameError etc*
  - *Explicitly state Exception name after the try keyword*
- The following code is better:

```
def myFunc(myKey):
    try:
        return(stateCapitals[myKey])
    except keyError:
        queryList.append(myKey)
        return(None)
```

Assertions: Another Way of Checking for Errors

- One way to check correctness of programs is to check if a given condition is true at a particular point in the code

```
def promptForIntInput():
    data = int(input("Give me an integer between 0 and 100 : "))
    assert data < 0 or data > 100, 'data is out of range'
    else:
        return data
try:
    myInput = promptForInInput()
except AssertionError as errorMessage:
    print(errorMessage)
```

Assertion and Exception are Useful to Handle IO Errors

```
import sys

def linux_interaction():
    assert 'linux' in sys.platform, "Function only runs on Linux."
    print('Doing something.')

try:
    linux_interaction()
    with open('file.log') as file:
        read_data = file.read()
except IOError as fnf_error:
    print(fnf_error)
except AssertionError as error:
    print(error)
    print('Linux linux_interaction() function was not executed')
```

Managing Files with `with`

```
myFile = open(”filename”,r)
for i in myFile.readlines():
    do something
myFile.close()
```

- Files are an external resource
  - *When you open a file, you must close it when you are done*
  - *What happens if there is an error while reading from a file? The code to close the file will never get executed*

```
with open(”filename”,r) as myFile:
    for i in myFile.readlines():
        do something
myFile.close() #ValueError: I/O operation on closed file.
```