MR2020: Coding for METOC

Module 9: Errors and Exceptions

**Many of the examples and explanations shown herein were generated by ChatGPT V4.0.*

What are errors and why do they occur?

Exceptions occur when Python cannot execute the code. They can occur for several different reasons. When learning Python, syntax errors are commonly raised. There are numerous types of errors that can be triggered (ask ChatGPT for a full list and you will get a long list of errors that you will mostly never encounter). Some of the commonly triggered built-in exceptions that you may encounter include

- SyntaxError
- IndentationError
- NameError
- IndexError
- TypeError
- ImportError
    - ModuleNotFoundError
- ZeroDivisionError
- AttributeError
- ValueError
- MemoryError

This module provides background on what each of these exceptions means, including an example of the error.

SyntaxError

Syntax errors occur when an incorrect statement is encountered during code execution. It is commonly caused by misspelling Python key words, leaving out colons, or incorrect usage of parenthesis, brackets, or braces.

```
# Syntax errors
impot numpy as np
```

```
    impot numpy as np
    ^
SyntaxError: invalid syntax
```

```
for i in range(0,4)
    A = 2
```

```
    for i in range(0,4)
    ^
SyntaxError: expected ':'
```

IndentationError

Python uses indentation to define blocks of code. Improper indentation will lead to an error. For example, text inside control flows or functions must be indented.

```
A = 2
if A > 2:
print('A is big!')
```

```
print('A is big!’)
^
IndentationError: expected an indented block after 'if' statement on line 2
```

Inconsistent indentation will also trigger an error.

```
A = 2
if A > 2:
    print('A is big!’)
   B = 3
```

```
B = 3
^
IndentationError: unindent does not match any outer indentation level
```

One line is tabbed and the other is indented with 3 spaces, creating inconsistent indentation.

NameError

A NameError occurs when the code attempts to use a variable or function name that has not been defined. This can happen when a variable name is misspelled and often happens when a variable is simply not defined at the point where the code encounters it.

```python
import numpy as np
avg = np.mean([4,3,2])
print(average)
```

```
          1 import numpy as np
          2 avg = np.mean([4,3,2])
---->     3 print(average)
NameError: name 'average' is not defined
```

Error occurs because the variable ′average′ has not been assigned. Only ‘avg′ would work.

```python
mylist = [4,3,2]
def calcavg(list):
    avg = np.mean(list)
    return avg
calcavg(mylist)
print(avg)
```

```
          4     return avg
          5 calcavg(mylist)
---->     6 print(avg)
NameError: name 'avg' is not defined
```

Error occurs because scope of variable ‘avg′ is limited to inside function and we are trying to print it outside the function.

IndexError

An IndexError occurs when the code attempts to access a position in a sequence (such as a list) that does not exist.

```
                  Element 0         Element 1
fruits = ["apple", "banana"]
print(fruits[2])
```

```
      1 fruits = ["apple", "banana"]
---->   2 print(fruits[2])
IndexError: list index out of range
```

In this example, there are only 2 elements in the list. Therefore, fruits[2], which attempts to access the 3rd element, throws an error.

TypeError

A TypeError occurs when you try to execute an illegal operation on a particular data type. For example, you can't add a string to an integer number. A TypeError can also occur when the incorrect data type is passed to a function that requires a specific data type.

```python
def divide(a, b):
    return a / b

# Trying to pass a string instead of a number
result = divide(10, "2")
print(result)
```

Suppose we have this code, which includes a function that requires two numbers to divide.

The error occurs at the line where we try to pass an int and str to the function and inside the function where the division is attempted.

```
          2     return a / b
          4 # Trying to pass a string instead of a number
----->    5 result = divide(10, "2")
          6 print(result)

line 2
          1 def divide(a, b):
----->    2     return a / b
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

TypeError

A TypeError occurs when you try to execute an illegal operation on a particular data type. For example, you can't add a string to an integer number. A TypeError can also occur when the incorrect data type is passed to a function that requires a specific data type.

```python
class Dog:
    def __init__(self, name):
        self.name = name

    def bark(self):
        return "Woof!"

# Create an instance of Dog
my_dog = Dog("Rex")

# Attempt to call name 
attribute as a method
my_dog.name()
```

A TypeError may also occur when erroneously trying to call a class attribute as a method. In the line

```python
my_dog.name()
```

the name attribute should be accessed without parentheses. Instead, it is being called as a method and the following error appears because you cannot call a string (which "Rex" is):

```
TypeError: 'str' object is
not callable
```

ImportError

An ImportError occurs when it attempts to load a module that is not accessible. Mostly commonly, this happens if the module name is misspelled or if it is not installed, and throws a ModuleNotFoundError.

Import errors can also occur when there are circular dependencies or syntax errors within a module that is being imported. These are less commonly encountered.

Example of simple typo in module import

```
----> 1 import numppy as np
ModuleNotFoundError: No module named 'numppy'
```

Example of attempting to import an actual module that is not installed.

```
----> 1 import pygrib as pg
ModuleNotFoundError: No module named 'pygrib'
```

ZeroDivisionError

When Python attempts to divide by zero, it returns a ZeroDivisionError.

```
A = [4,3,1]
B = [0,3,2]

for i,j in zip(A,B):
    print(i/j)
```

```
      2 B = [0,3,2]

      4 for i,j in zip(A,B):
----> 5 print(i/j)
ZeroDivisionError: division by zero
```

NOTE: Below code does not return error. By default, NumPy returns inf or –inf when dividing by zero. However, it will print out a warning that does not stop the code.

```
A = np.array([4,3,1])
B = np.array([0,3,2])
print(A/B)
```

Returns

```
RuntimeWarning: divide by zero
encountered in divide A/B

array([inf, 1. , 0.5])
```

AttributeError

An AttributeError occurs when you try to access or call an attribute (such as a method or property) that does not exist on an object. This typically happens when there is a typo, a misunderstanding of an object's available attributes, or a misuse of a method or property.

```python
class Pair:
    def __init__(self,num1,num2):
        self.number1 = num1
        self.number2 = num2
    def sum(self):
        return self.number1 + self.number2
    def product(self):
        return self.number1 * self.number2
```

```python
nums = Pair(3,4)
nums.product()   # OK
nums.num1        # Error
nums.compare()   # Error
```

```
AttributeError: 'Pair' object has no
attribute 'num1'
```

```
AttributeError: 'Pair' object has no
attribute 'compare'
```

ValueError

A ValueError occurs in Python when a built-in operation or function receives an argument that has the right type but an inappropriate value. This often happens when the input does not conform to the expected domain or range of the function.

Common Causes of ValueError

1. Invalid Literal for Conversion

Attempting to convert a string to an integer or float when the string contains non-numeric characters.

2. Out-of-Range Values

Passing a number that is outside the valid range for a given function or method.

3. Incorrect Data Format

Providing data in an incorrect format that a function cannot process.

4. Mismatch in Data Length

Supplying data sequences of mismatched lengths when a specific length is required.

```
# Attempt to cast string as int
number = int("abc123")
```

```
ValueError: invalid literal for
int() with base 10: 'abc123'
```

```
# Attempting to access a list
element with an invalid index
values = [10, 20, 30]
value = values.index(40)
```

```
ValueError: 40 is not in list
```

MemoryError

A MemoryError occurs in Python when the program runs out of memory, typically because the memory required for an operation exceeds what is available on the machine. This often happens when dealing with very large data structures, inefficient data handling, or infinite loops that consume memory.

Common Causes of MemoryError

1. Excessive Memory Allocation

Creating excessively large data structures such as lists, dictionaries, or NumPy arrays that the system cannot handle.

2. Infinite Loops with Accumulating Data

Loops that continuously append data to a collection without a termination condition.

3. Processing Large Files or Data Sets

Reading large files or datasets into memory all at once rather than using efficient data processing techniques.

4. Inefficient Data Structures

Using data structures that are not optimized for memory usage, leading to excessive consumption.

Handling Exceptions Explicitly

There may be times when the programmer wants to raise an error that is not normally raised during code execution.

For example, suppose you have a list of dictionaries containing information about several people, including age. Perhaps you want to calculate the average age of the people, but you want to ensure that an error is raised if anyone's age is less than 0.

```
people = [
{'name': 'Bill', 'age': 35},
{'name': 'Sue', 'age': 52},
{'name': 'Mike', 'age': 23},
{'name': 'Zoe', 'age': -6},
{'name': 'Peter', 'age': 14}
]

total = 0
    for person in people:
        age = person['age']
        # Explicitly raise error where it wouldn't otherwise occur
        if age < 0:
            raise ValueError('No negative ages allowed.')
        else: total += age
```

In this example, Python would happily calculate the average of the 5 numbers, but since negative ages don't make sense, we need to explicitly call an error. A ValueError makes sense to raise here, but we could define a custom error class.

The `raise` keyword calls an error explicitly in Python.