COMP642

Object Oriented Programming

Lectorial 9 - Exception, Defining Own Exceptions

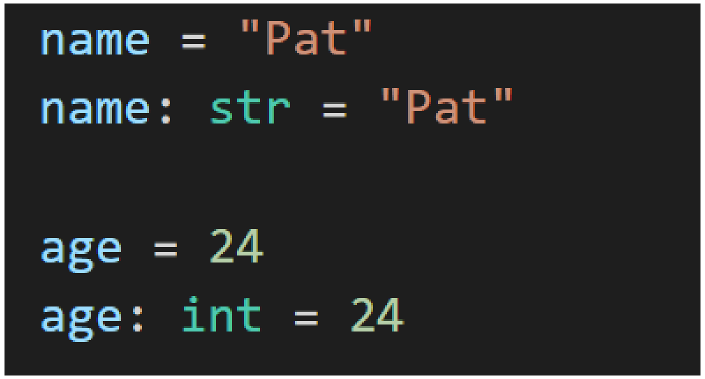
What are Type Hints?

- Python has dynamic types, which increases debugging time and makes the behaviour of variables and methods unpredictable.
- Don’t have to rigorously define and track variable types.
- Useful to know which object types are associated with which variables.
- Type hinting – formal solution to statically indicate the type of value within a Python code.
- Specified in Python Enhancement Proposal 484 (PEP 484) and introduced in Python 3.5.
- Improve code quality.

How Python Uses Type Hints?

- For developer and not for runtime.
- No type checking happens at runtime.
- Assumes the existence of a separate off-line type checker which users can run over their source code voluntarily.
- Provide developers a way to make their code as self-describing as possible.

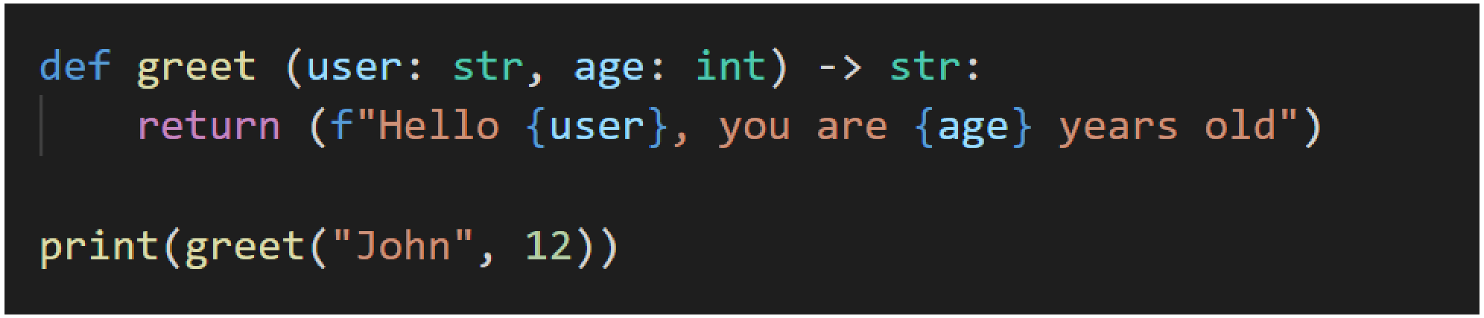
The Syntax (1)



```python
name = "Pat"
name: str = "Pat"

age = 24
age: int = 24
```

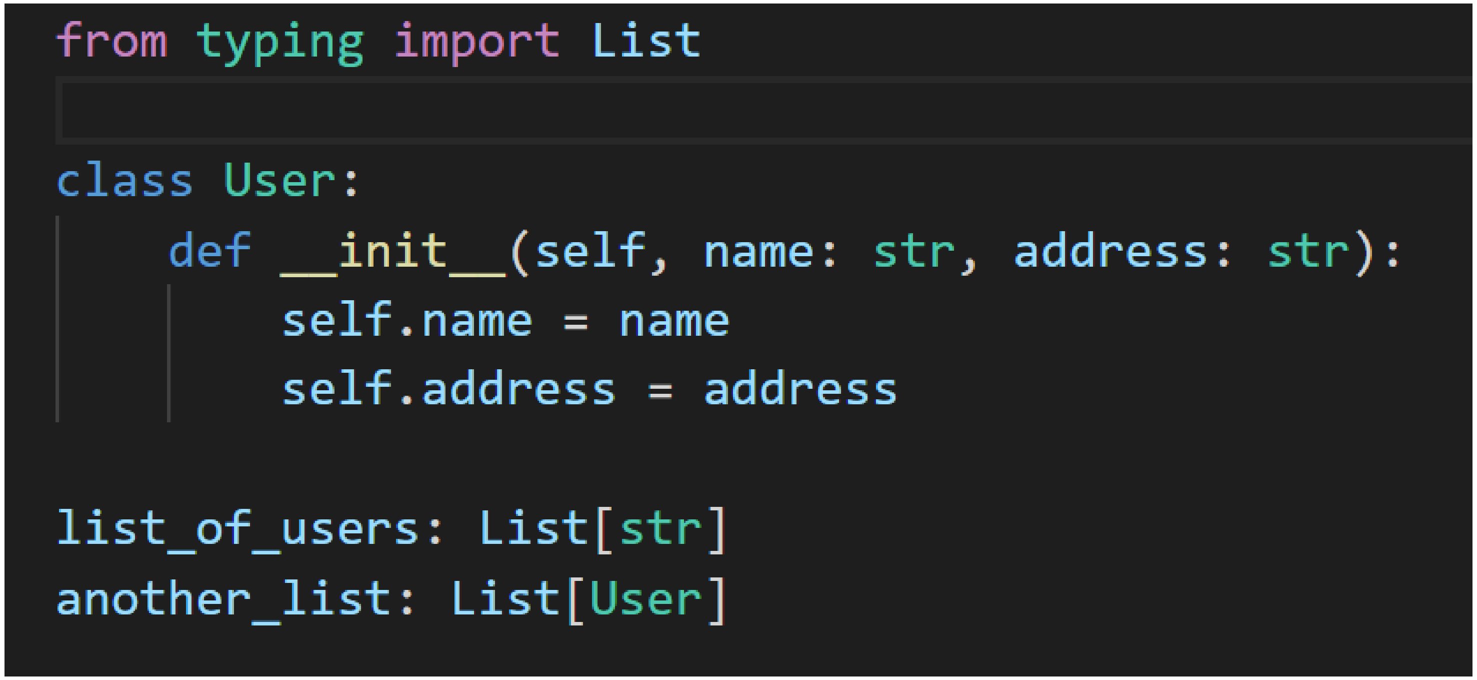
Declaring variables with type hints

The Syntax (2)



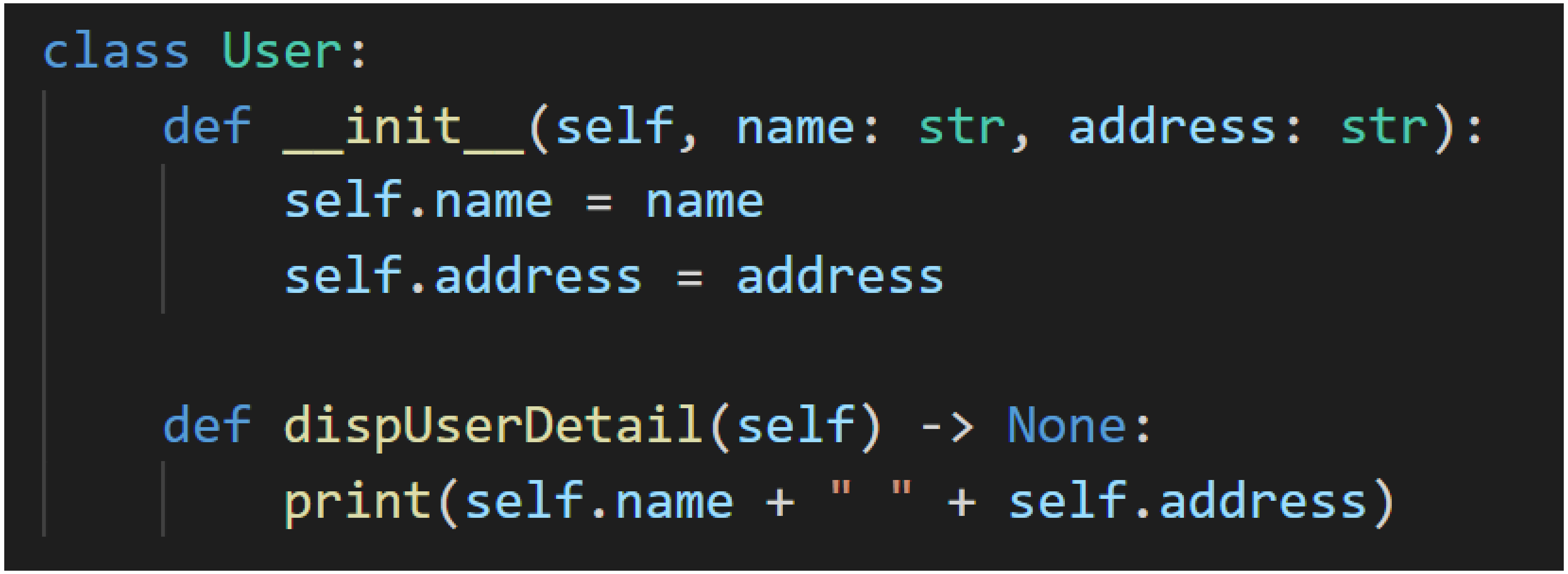
```python
def greet (user: str, age: int) -> str:
    return (f"Hello {user}, you are {age} years old")

print(greet("John", 12))
```

The Syntax (3)



```
from typing import List

class User:
    def __init__(self, name: str, address: str):
        self.name = name
        self.address = address

list_of_users: List[str]
another_list: List[User]
```

The Syntax (4)



```python
class User:
    def __init__(self, name: str, address: str):
        self.name = name
        self.address = address

    def dispUserDetail(self) -> None:
        print(self.name + " " + self.address)
```

Recommendations

- Use normal rules for colons; no space before and one space after a colon (text: str).
- Use spaces around the = when combining an argument annotation with a default value (flag: bool = True).
- Use spaces around the -> arrow (def greet(...) -> str).
- To enforce type checking, we can use a static type checker such as “mypy”.

What is an Exception?

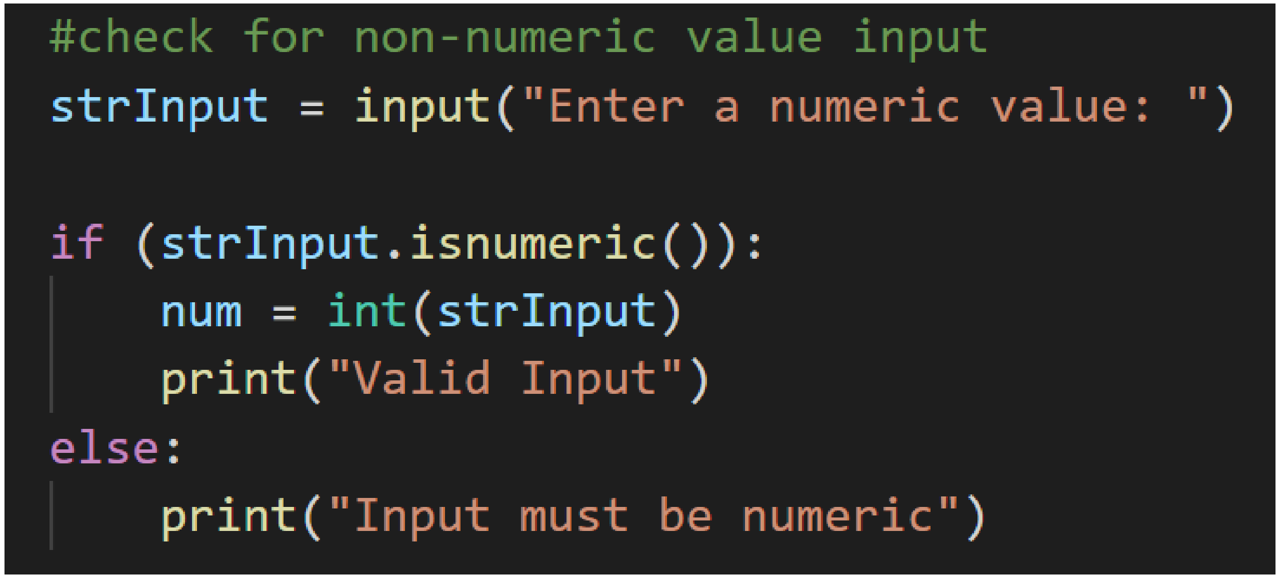
an **event which** occurs during the execution of a program that disrupts the normal flow of the programs' instruction

- Errors that occur at runtime.
- Are not unconditionally fatal.
- E.g., **name error, division by zero, file not found, type error**.
- A syntax error occurs when the parser detects an incorrect statement, an exception occurs whenever syntactically correct Python code results in an error.

Checking Data Entry *without* using Exceptions

- E.g., check for non-numeric value in a text box

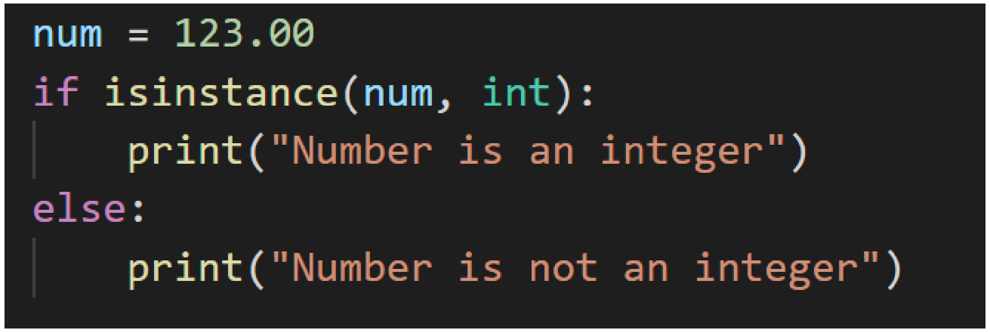
– could test for this in the program



```
#check for non-numeric value input
strInput = input("Enter a numeric value: ")

if (strInput.isnumeric()):
    num = int(strInput)
    print("Valid Input")
else:
    print("Input must be numeric")
```

...More Checking

- E.g., could also do a check to see if the number is a whole number (integer)

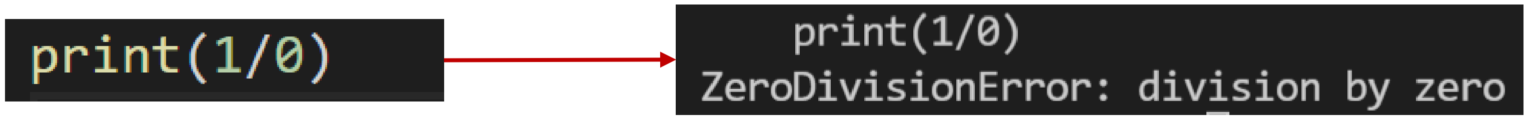


```python
num = 123.00
if isinstance(num, int):
    print("Number is an integer")
else:
    print("Number is not an integer")
```

- Easier to use the exception mechanism at the time when the string is converted to integer.
- Traditionally we used LBYL (look before you leap), but Python developers use EAFP (easier to ask for forgiveness than permission) using try...exception block.

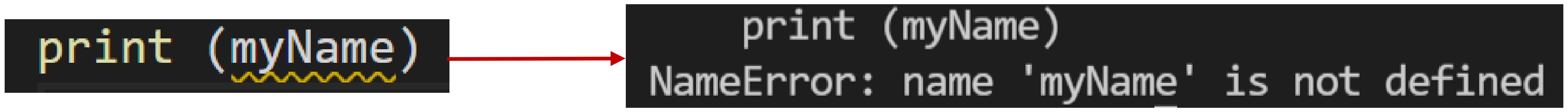
Exceptions

- Errors detected during execution are called exceptions.
- Not fatal – can be handled in Python programs.
- Each time an error is detected in a program, the Python interpreter raises (throws) an exception.



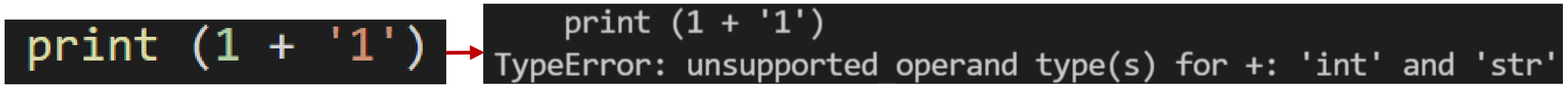
```
print(1/0)
```

```
    print(1/0)
ZeroDivisionError: division by zero
```



```
print (myName)
```

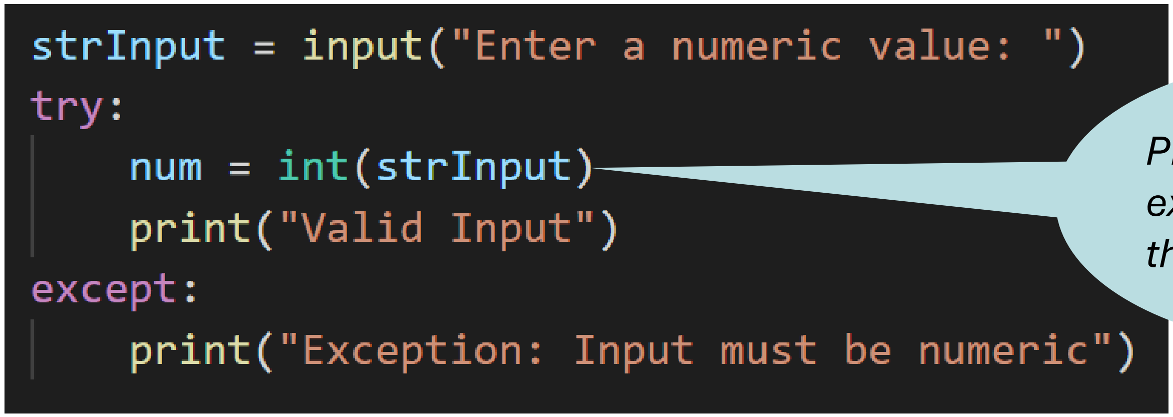
```
    print (myName)
NameError: name 'myName' is not defined
```



```
print (1 + '1')
```

```
    print (1 + '1')
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Checking Data Entry using Exceptions



```
strInput = input("Enter a numeric value: ")
try:
    num = int(strInput)
    print("Valid Input")
except:
    print("Exception: Input must be numeric")
```

Program will execute this, unless there is a problem.

- As soon as an error is detected in the **try** block, control is transferred to the **exception** block where it is handled.

How are Exceptions Handled?

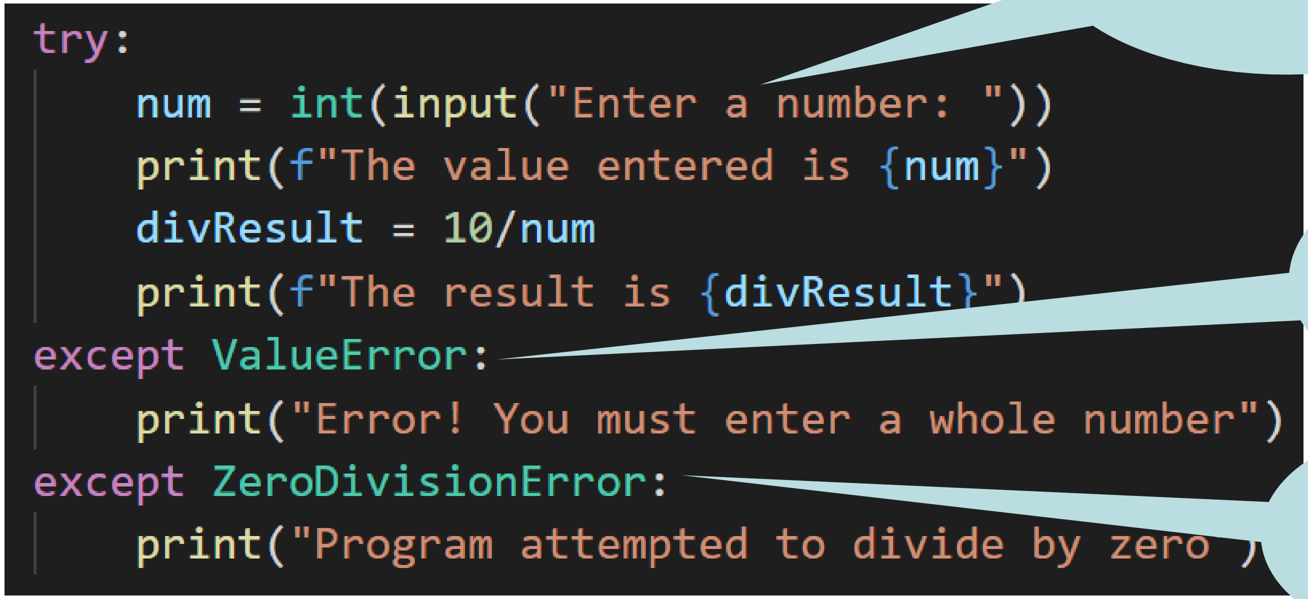
1. The try clause is executed.
2. If no exception occurs, the except clause is skipped and execution of try statement is finished.
3. If an exception occurs during the execution of try clause, the rest of the clause is skipped and the except clause is executed.

Common Exception Types

ValueError	a function gets an argument of correct type but improper value
AttributeError	attribute assignment or reference fails
EOFError	input() function hits EOF condition
NameError	variable is not found in local or global scope
OverflowError	result of an arithmetic operation is too large to be represented
TypeError	a function or operation is applied to ab object of incorrect type
ZeroDivisionError	Second operand of division or module operation is zero

Catching Particular Exception Types

Can we control what happens, depending on the exception type we might get?



```python
try:
    num = int(input("Enter a number: "))
    print(f"The value entered is {num}")
    divResult = 10/num
    print(f"The result is {divResult}")
except ValueError:
    print("Error! You must enter a whole number")
except ZeroDivisionError:
    print("Program attempted to divide by zero")
```

In case of error in the try block...

...control could be transferred to here

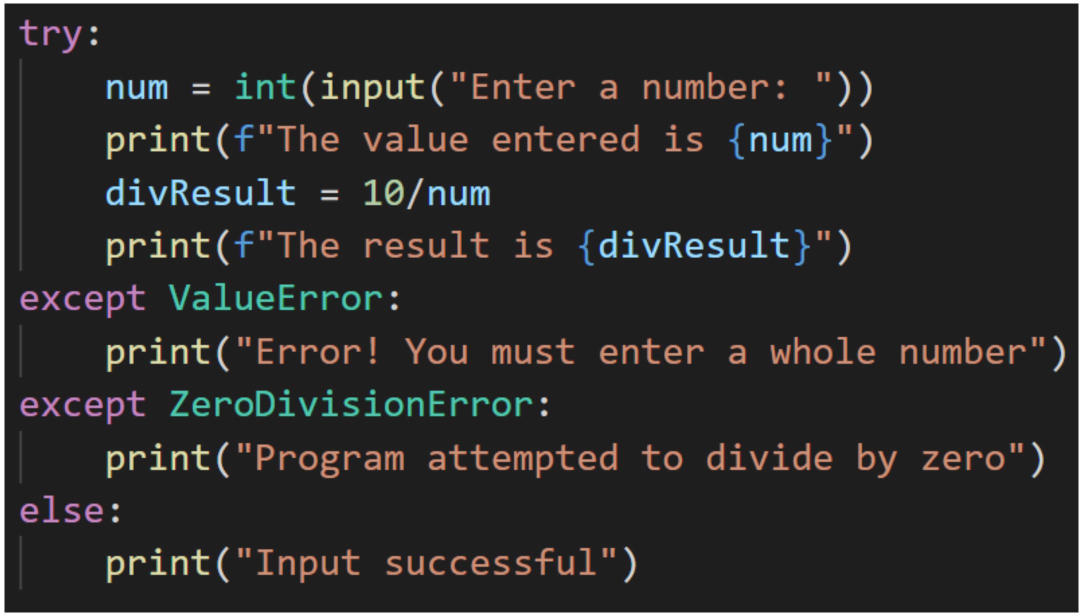
...or to here

General Structure

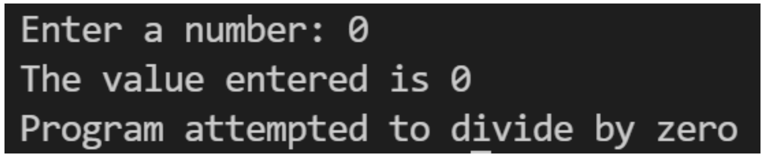
```
try:
        normal code
exception ExceptionType1:
        exception handling code
exception ExceptionType2:
        exception handling code
……
else:
        program code to run if “try” block does not encounter any error
finally:
        clean-up code – runs regardless of errors in the “try” or “else”
```

try-except-else

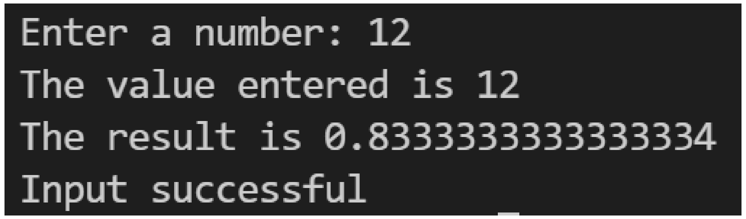
- The **else** block runs if there are no exceptions raised from the “try” block.



```python
try:
    num = int(input("Enter a number: "))
    print(f"The value entered is {num}")
    divResult = 10/num
    print(f"The result is {divResult}")
except ValueError:
    print("Error! You must enter a whole number")
except ZeroDivisionError:
    print("Program attempted to divide by zero")
else:
    print("Input successful")
```



```
Enter a number: 0
The value entered is 0
Program attempted to divide by zero
```



```
Enter a number: 12
The value entered is 12
The result is 0.8333333333333334
Input successful
```

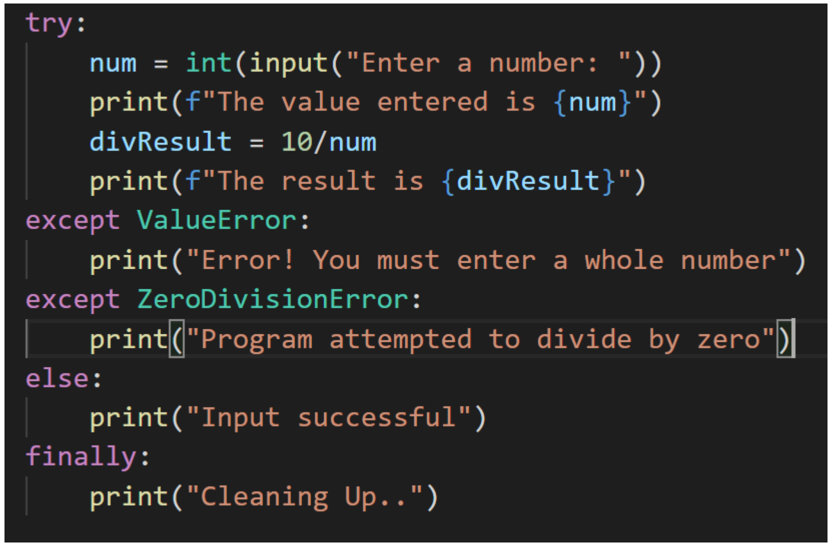
Clean-up Code (finally)

- The **finally** block is:

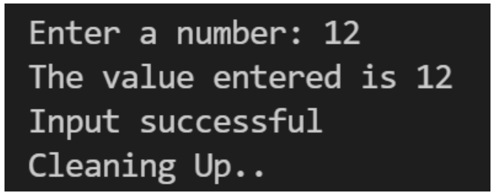
– always executed **after** the **try** block and any **except** block that is executed as a result of an exception.

– used to do tasks such as closing files that were opened during the try block.

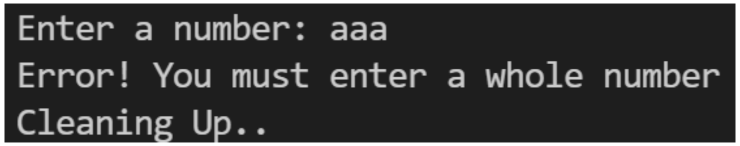
– not required (optional).



```python
try:
    num = int(input("Enter a number: "))
    print(f"The value entered is {num}")
    divResult = 10/num
    print(f"The result is {divResult}")
except ValueError:
    print("Error! You must enter a whole number")
except ZeroDivisionError:
    print("Program attempted to divide by zero")
else:
    print("Input successful")
finally:
    print("Cleaning Up..")
```



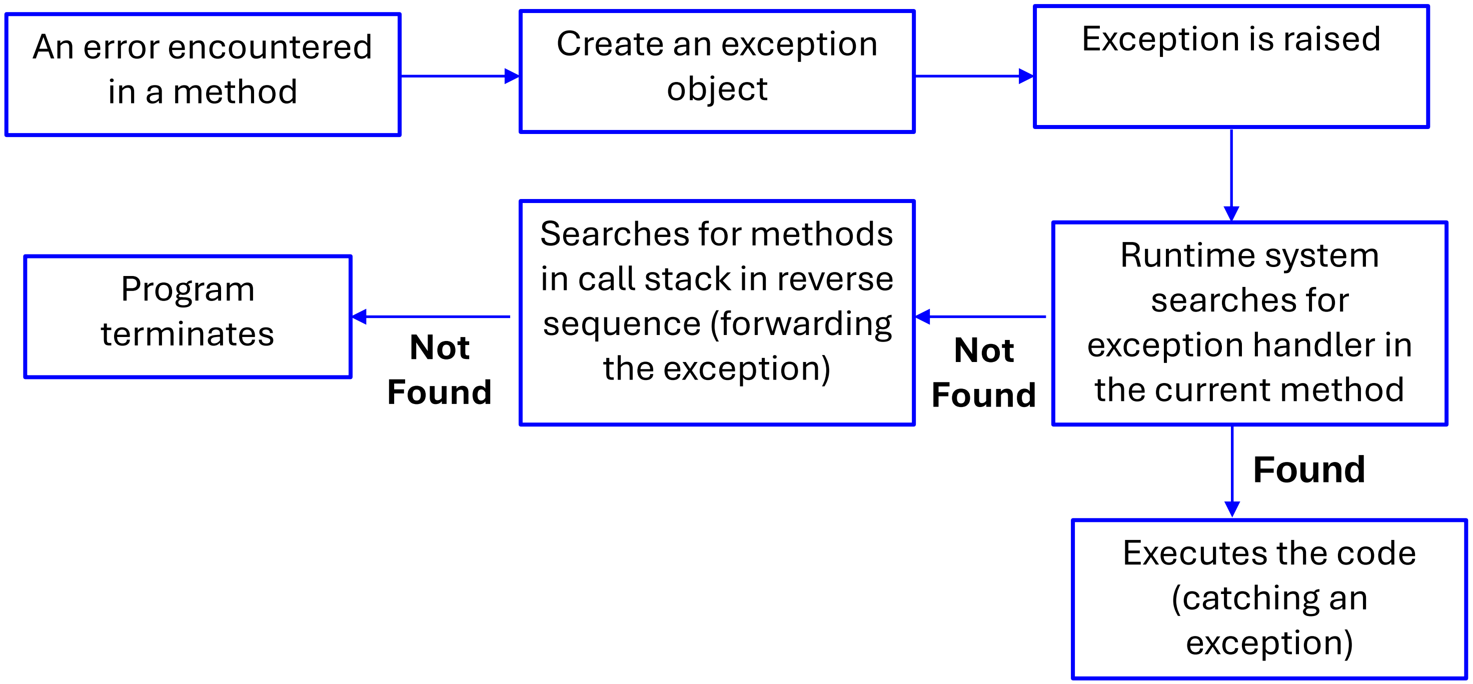
```
Enter a number: 12
The value entered is 12
Input successful
Cleaning Up..
```



```
Enter a number: aaa
Error! You must enter a whole number
Cleaning Up..
```

Python Exception Handling

- When an error occurs, Python interpreter creates an object called the exception object.
- This object contains information about the error (type, file name, where the error occurred).


An error encountered in a method
Create an exception object
Exception is raised
Runtime system searches for exception handler in the current method
Found
Executes the code (catching an exception)
Not Found
Searches for methods in call stack in reverse sequence (forwarding the exception)
Not Found
Program terminates

Reporting Errors from within a Class

- don’t want to use print a message from within a class method?

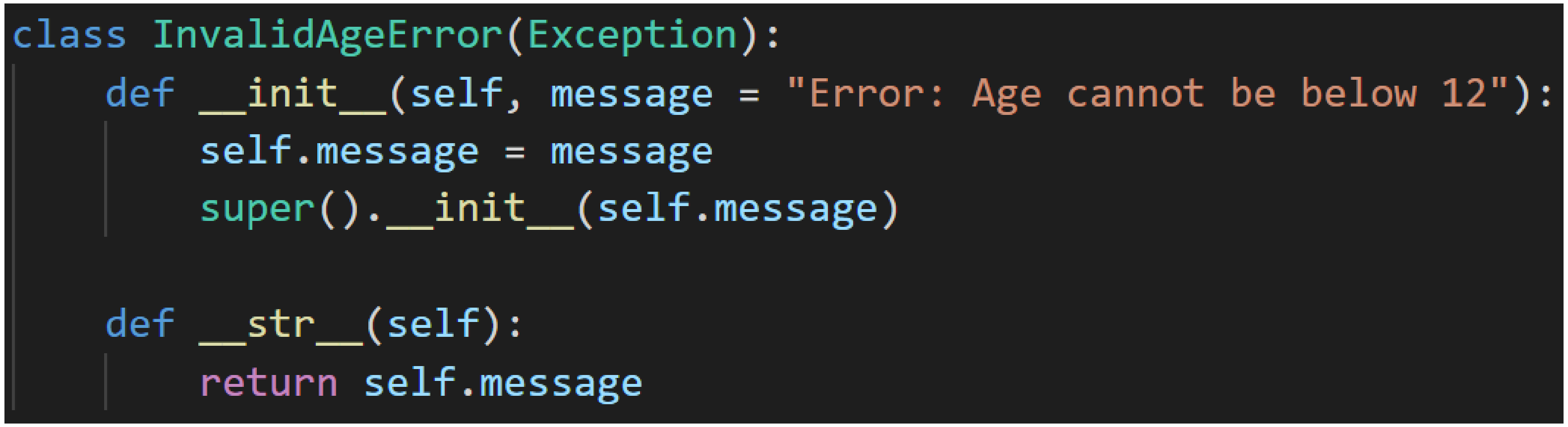
MVC! Remember to keep View (buttons, msgboxes, etc) separate from Model (classes)

- could have a method ***CheckError*** that will return an error code indicating if error occurred during last method call.
- better alternative is to **throw** an exception for programmer-defined exception class.

Programmer-defined Exceptions

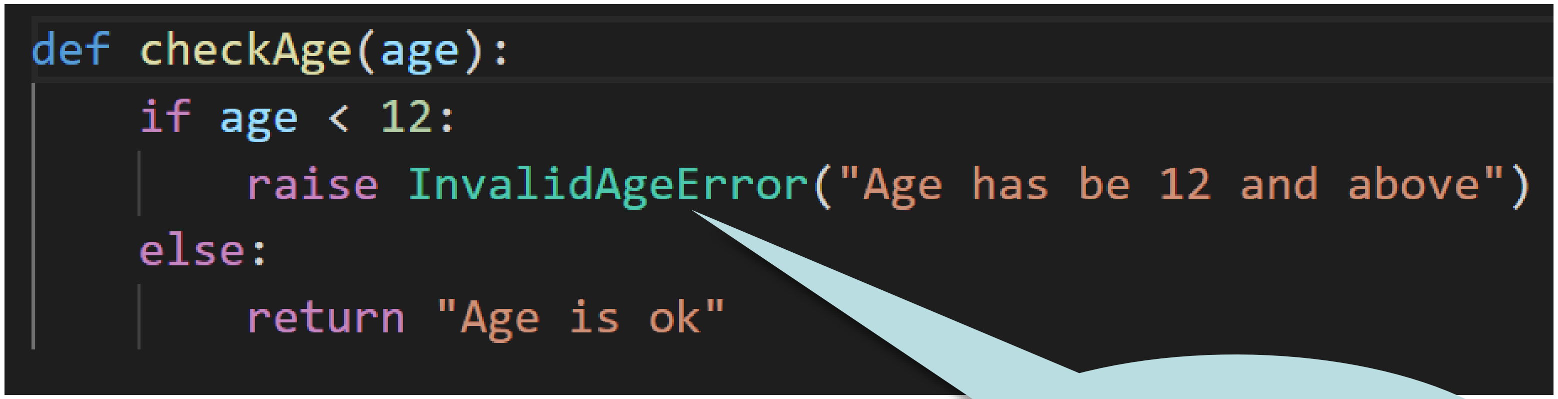
- define your own exception class that inherits from Exception.
- Most exceptions are named as names than end in “Error”.
- create a constructor that accepts a string ***message*** as an argument and passes it to the base constructor.
- in the code that detects the error, ***raise*** the exception including a meaningful message.
- place that method call in a ***try…execute***.

Defining Your Exception Class...



```python
class InvalidAgeError(Exception):
    def __init__(self, message = "Error: Age cannot be below 12"):
        self.message = message
        super().__init__(self.message)

    def __str__(self):
        return self.message
```

...Generating the Exception...

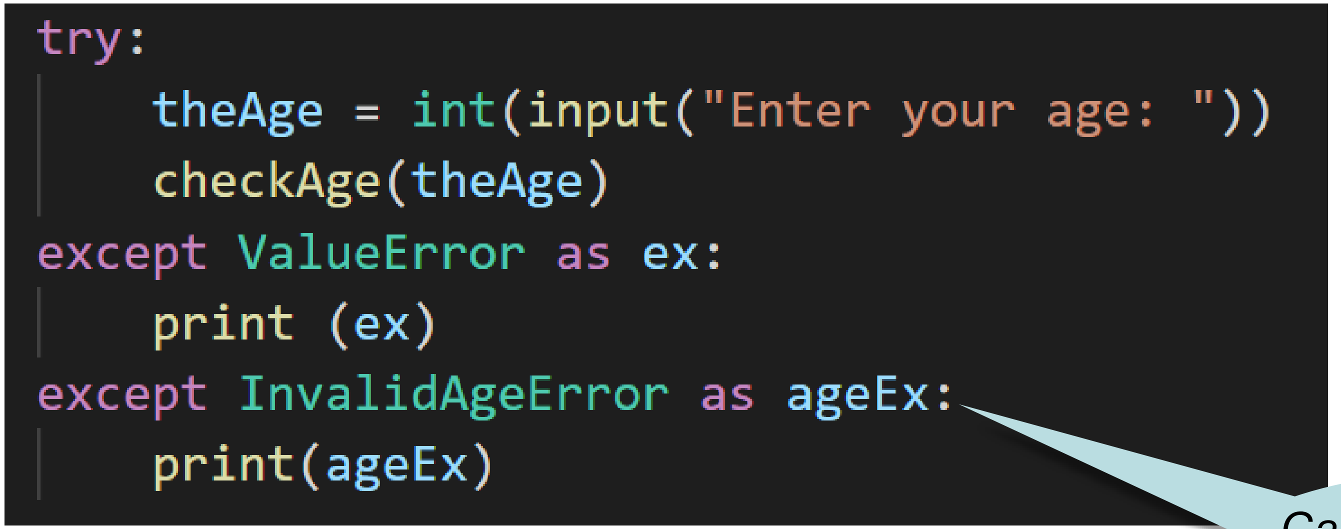


```
def checkAge(age):
    if age < 12:
        raise InvalidAgeError("Age has be 12 and above")
    else:
        return "Age is ok"
```

Create the new exception object and explicitly throw it

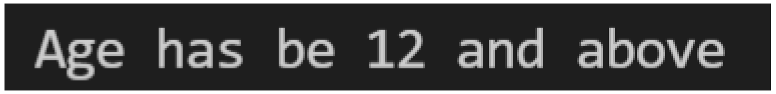
- The **raise** keyword causes an exception of the given class to occur.

...Catching the Exception (1)



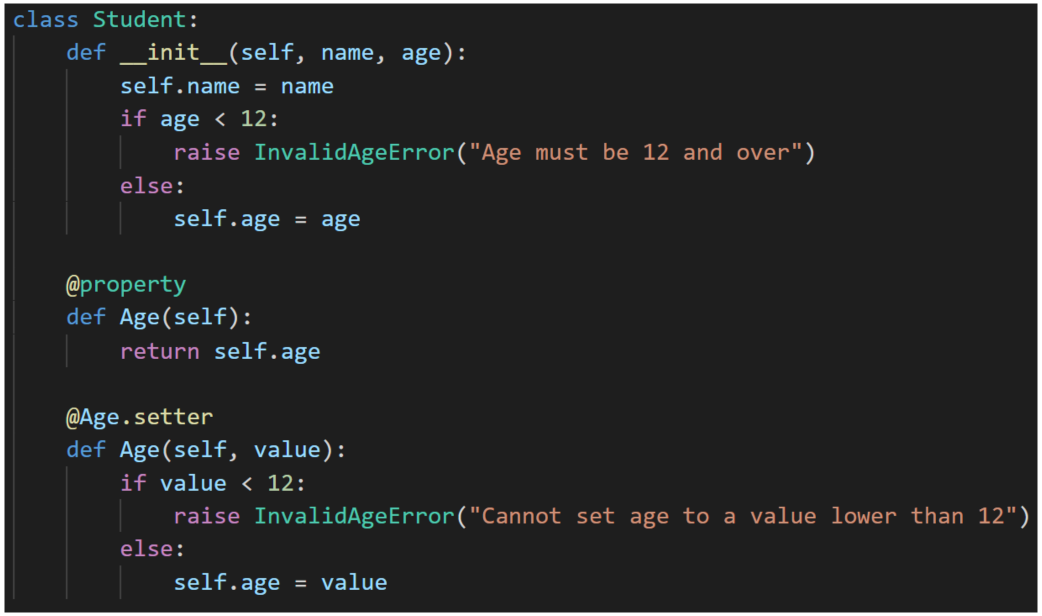
```
try:
    theAge = int(input("Enter your age: "))
    checkAge(theAge)
except ValueError as ex:
    print (ex)
except InvalidAgeError as ageEx:
    print(ageEx)
```

Catching the specific exception

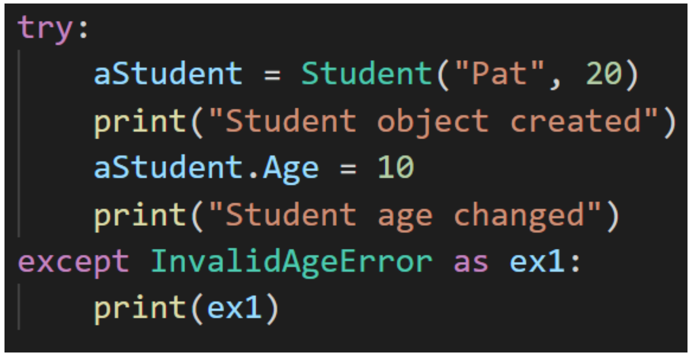


```
Age has be 12 and above
```

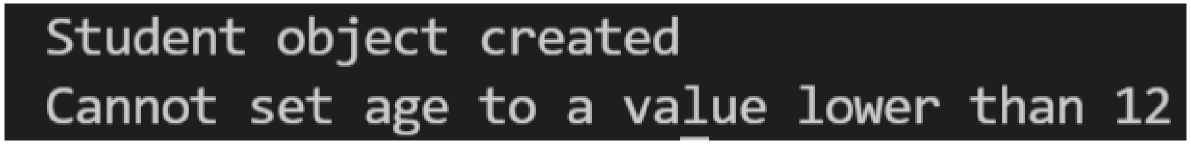
...Catching the Exception (2)



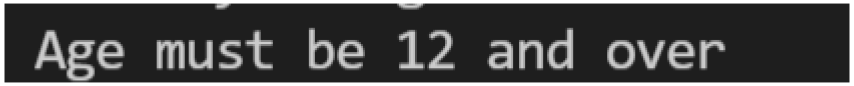
```python
class Student:
    def __init__(self, name, age):
        self.name = name
        if age < 12:
            raise InvalidAgeError("Age must be 12 and over")
        else:
            self.age = age

    @property
    def Age(self):
        return self.age

    @Age.setter
    def Age(self, value):
        if value < 12:
            raise InvalidAgeError("Cannot set age to a value lower than 12")
        else:
            self.age = value
```



```python
try:
    aStudent = Student("Pat", 20)
    print("Student object created")
    aStudent.Age = 10
    print("Student age changed")
except InvalidAgeError as ex1:
    print(ex1)
```



```
Student object created
Cannot set age to a value lower than 12
```

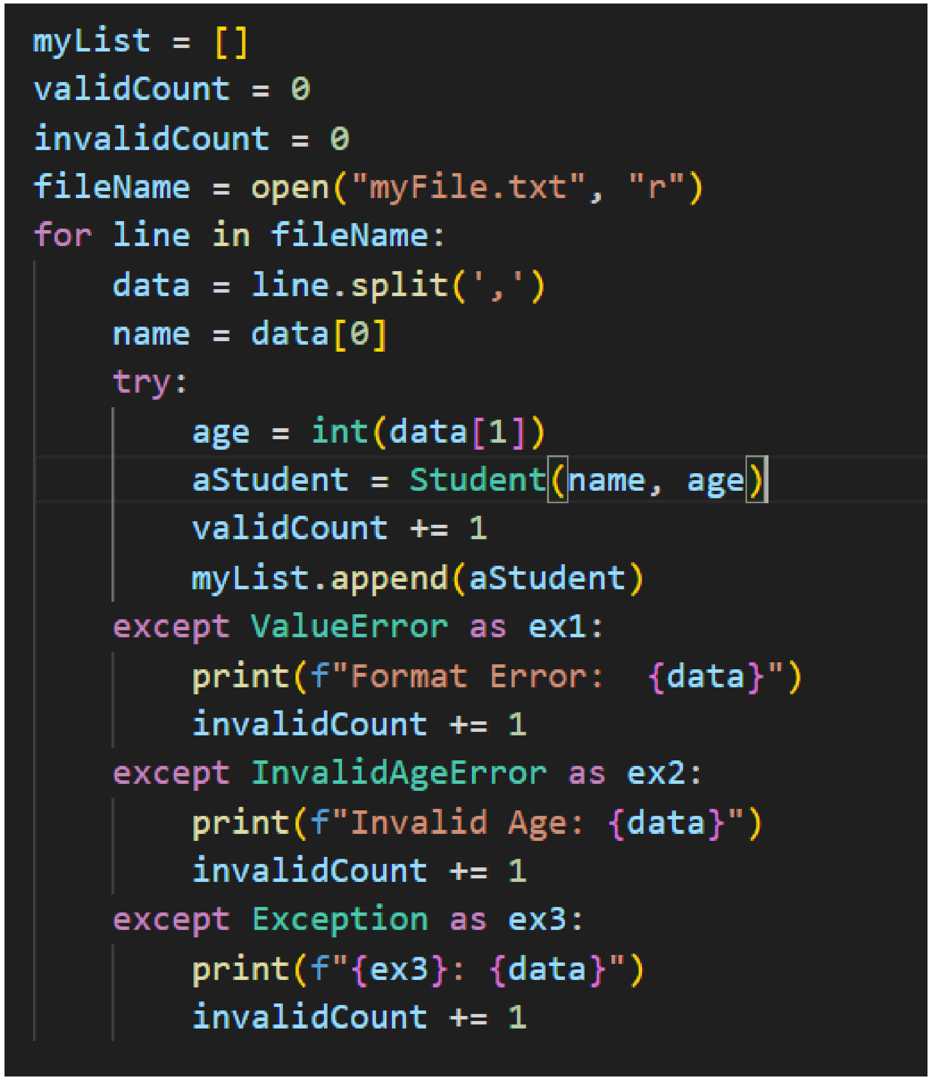


```
Age must be 12 and over
```

Processing Input from a File

- If we are creating a List of objects from data in a **file**, we:
  - might not want to halt processing if an error occurs.
  - need to report errors (eg, in a list box or log file).
  - need to identify which line(s) contain errors.
  - need to make sure that objects aren't created for lines that contain errors.

Processing Input from a File



```python
myList = []
validCount = 0
invalidCount = 0
fileName = open("myFile.txt", "r")
for line in fileName:
    data = line.split(',')
    name = data[0]
    try:
        age = int(data[1])
        aStudent = Student(name, age)
        validCount += 1
        myList.append(aStudent)
    except ValueError as ex1:
        print(f"Format Error:  {data}")
        invalidCount += 1
    except InvalidAgeError as ex2:
        print(f"Invalid Age: {data}")
        invalidCount += 1
    except Exception as ex3:
        print(f"{ex3}: {data}")
        invalidCount += 1
```