Digital Career Institute

Basics - Exceptions

Digital Career Institute

Goal of the Submodule

The goal of this submodule is to help understand and use exceptions (errors) in Python. By the end of this submodule, the learners should be able to understand how-to:

- Use Casting and Conversion
- *Throw* and *Catch* exceptions
- Create *custom* exceptions
- *Raise* an exception
- Use *built-in exceptions* effectively to build robust Python programs

Topics

- **What are casting and conversion?**
  - Why use explicit conversion?
- **What are exceptions?**
  - Why use exceptions?
- **Errors & exceptions**
  - Syntax errors
  - Logical errors(exceptions)
- **The `try` and `except` blocks to handle exceptions**
  - Catching specific exceptions
  - Handle multiple exceptions with a single except clause
- **Using `try` with `finally`**
- **Raising exceptions**
- **Exception classes: custom exceptions**
  - Customizing exception classes
- **Built-in exceptions**
  - Exception class hierarchy

Casting and Conversion

Conversion in Python

Digital Career Institute

- Type casting is basically type conversion => Change a Python object from one type to another.

- There are 2 types of conversion:

    - Implicit type conversion.
    - Explicit type conversion, also called **type casting.**

Implicit Conversion

- Implicit conversion happens when Python converts types without a clear instruction to do so in the code. It's done automatically.

```
a = 1
b = 2
pi = 3.14

print(a/b)
#>>> 0.5

print(b/a)
#>>> 2.0

print(a + pi)
#>>> 4.14

c = "Hello"
d = (1, 2, 3)

print(c + d)
#>>> Error
```

Implicit conversion

Python automatically converted the numbers to float when it was needed.

Python however doesn't automatically convert strings to tuples.
Solution: **Type Casting** (explicit conversion).

Explicit Conversion: Type Casting

- The user indicates the conversion.
- It's done by calling one of the functions **tuple(), list(), int(), str(), ...**

```
c = "Hello"
d = (1, 2, 3)
print(tuple(c) + d)
#>>> ('H', 'e', 'l',
'l', 'o', 1, 2, 3)


m = 120
print(list(m))
#>>> TypeError:
'int' object is not
iterable
```

Type Casting

`tuple(c)` converted `c` from string to tuple.

Not every type casting is possible. Only iterables can be cast into other iterables.

TypeError and ValueError

When casting in Python, two types of errors can happen:

TypeError:

When the conversion is impossible between the two types indicated.

Example:
a = (1, 2, 3)
b = int(a) #>> generates an error.

Because the type `tuple` cannot be cast to `int`.

ValueError:

When the conversion is possible between the two types, but the value is wrong and can't be converted.

Example:
int("34") #>> Works well. It casts string to int.
int("Moi") #>> generates a ValueError.

Because the value "`Moi`" can't be cast to an `int`.

We learned ...

- That, most Python objects can be converted from one type to another.
- That implicit conversion is done by Python automatically when needed.
- That explicit conversion (type casting), is done by the user in the code.
- That not all type conversions are possible. `ValueError` or `TypeError` will be generated if the conversion is wrong.

Glossary

Term	Definition
Exception	An event that occurs during the execution of programs that disrupts the normal flow of execution
Error	An error is an action that is incorrect or inaccurate
Built-in method	A ready made functions to be used in Python

What are exceptions?

Introducing Exceptions

Digital Career Institute

An **exception** in python is an event that occurs during the execution of programs that disrupt the normal flow of execution.

It is an **object** (derived from `BaseException` class) that represents an error.

Exception objects contain:

- Error type (exception name)
- The state of the program when the error occurred
- An error message describing the error event

Exceptions exist to help developers **debug** their programs for errors.

Why use exceptions?

Digital Career Institute

Standardized error handling: Using built-in exceptions or creating a custom exception with a more precise name and description, one can adequately define the error event, which helps to debug the error event.

Cleaner code: Exceptions separate the error-handling code from regular code, which helps to maintain large code easily.

Robust application: With the help of exceptions, we can develop a solid application, which can handle error event efficiently.

Exceptions propagation: By default, the exception propagates the call stack if one doesn't catch it.

Different error types: Either you can use built-in exception or create your custom exception and group them by their generalized parent class, or differentiate errors by their actual `class`.

Standard exceptions

Digital Career Institute

Can you try to guess some common types of exceptions?

Standard exceptions

Digital Career Institute

Can you try to guess some common types of exceptions? Here are some!

A few standard exceptions:

- `FileNotFoundException`
- `ImportError`
- `RuntimeError`
- `NameError`
- `TypeError`

In Python, we can throw an exception in the `try` block and `catch` it in except block.

Errors & Exceptions

Recapping Errors

Digital Career Institute

An **error** is an action that is incorrect or inaccurate.

For example, a mistake in the program's syntax can cause an **error** due to which the program fails to execute.

Program errors can be broadly classified into two categories:

Syntax errors

Logical errors (or **exceptions**)

Syntax errors

Syntax errors occur when we are not following the proper **structure** or **syntax** of Python.

Syntax errors are also known as parsing **errors**.

Can you guess some basic syntax error cases?

Syntax errors

Syntax errors occur when we are not following the proper **structure** or **syntax** of Python.

Syntax errors are also known as parsing **errors**.

Some common Python syntax errors:

- Incorrect indentation
- Missing colon, comma, or brackets
- Putting keywords in the wrong place

Example

```
print("Welcome to DCI")
    print("Learn Python..")
```

Output

```
print("Learn Python..")
    ^
IndentationError: unexpected
indent
```

Logical errors

Even if a statement or expression is syntactically correct, the error that occurs at runtime is known as a **Logical error or Exceptions**.

In other words, **Errors detected during execution are called exceptions**.

Some common Python logical errors:

- Indenting a block to the wrong level
- Using the wrong variable name
- Making a mistake in a boolean expression

Example

```
x = 10
y = 0
z = x/y

print(z)
```

Output Exception

```
Traceback (most recent call
last):
  File "exception.py", line
3, in <module>
    z = x / y
ZeroDivisionError: division
by zero
```

Reading an exception

Python can trace back to the origin of exception.

Filename, line number, and module where the exception occured.

```
Traceback (most recent call last):
  File "exception.py", line 3, in <module>
    c = a / b
ZeroDivisionError: division by zero
```

Name of the exception.

Code which lead to the error.

`try`, `except` and `else` blocks to handle exceptions

`Try` and `Except`

Digital Career Institute

To **handle** exceptions we need to use `try` and `except` “blocks”.

*“A block is **a piece of Python program text that is executed as a unit.** The following are blocks: a module, a function body, and a class definition. Each command typed interactively is a block.”* [1]

Code that can raise an exception should be defined inside the `try` block and corresponding handling code inside the `except` block.

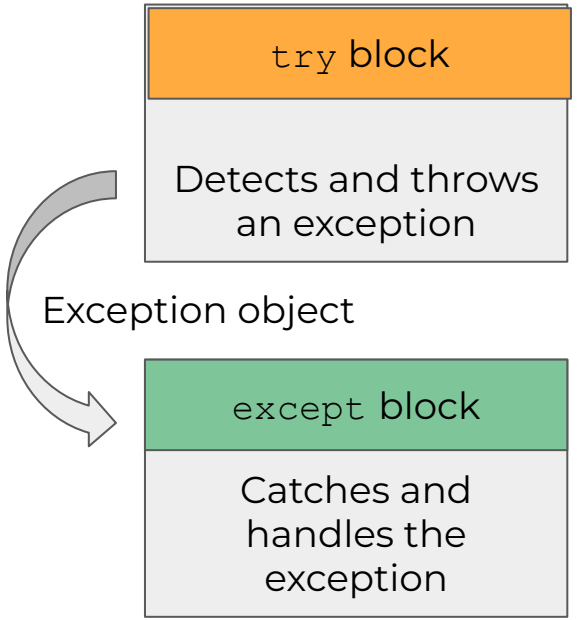
[1] https://docs.python.org/3/reference/executionmodel.html

try-except block

Syntax

```
try:
    # statement in try

except:
    # executed when
exception occured  in try
block
else:
    # executed if no
exception occured
```



Example

Let's take the code from the example in the errors section above.

Example without `try-except`

```
x = 10
y = 0
z = x/y

print(z)
```

Output Exception

```
Traceback (most recent call
last):
  File "exception.py", line
3, in <module>
    z = x / y
ZeroDivisionError: division
by zero
```

Example with `try-except`

```
x = 10
y = 0
try:
    z = x/y
    print("Answer:", z)
except:
    print("Cannot divide
value with zero.")
```

Output Exception

```
Cannot divide value with
zero.
```

Catching specific exceptions

Digital Career Institute

In the previous example, we did not catch a specific type of exception.

It is considered **good-practice** to specify exact type of exceptions the `except` block should catch.

For example, we can use `ValueError to` catch an exception that occurs when the user enters a non-numerical value instead of a number.

Let’s try this in our running example code.

Example

Example with `try-except`

```
x = "a"
y = 0
try:
    z = x/y
except ValueError:
    print("Entered value
is non-numeric.")
except ZeroDivisionError:
    print("Can't divide
value with zero.")
except TypeError:
    print("Cannot divide a
letter by a number")
else:
    print("Answer:", z)
```

Output Exception

```
Entered value is non-numeric.
```

It is important to note that the `try` block can be followed by multiple `except` blocks to handle different exceptions. **But only one exception will be executed when an exception occurs.**

Multiple exceptions in a single line

We can also handle multiple exceptions with a single `except` block, which can help reduce code repetition.

This can be done using `tuples` of values to specify multiple exceptions in an `except` clause.

Example with `try-except`

```
try:
    x = "a"
    y = 0
    z = x/y
    print("Answer:", z)
except (ValueError,
ZeroDivisionError,
TypeError):
    print("Entered value
throwing an error.")
```

Output Exception

```
Entered value is throwing an
error.
```

Using `try` and `finally`

`finally` block

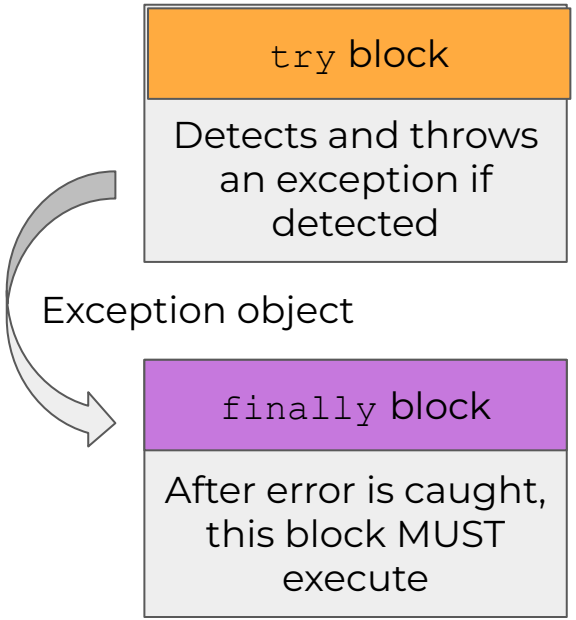
In Python, we have the `finally` block that **must** execute whether the try block raises an error or not.

Syntax

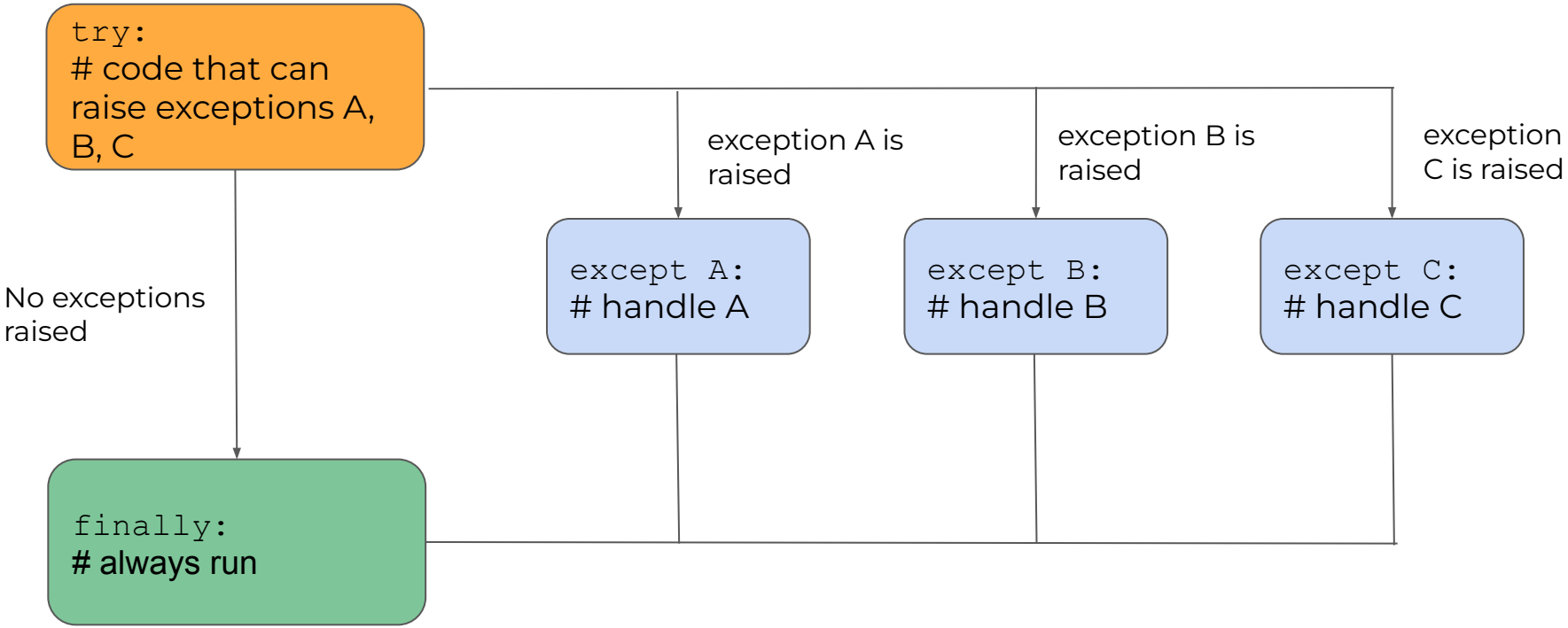
```
try:
    # block of code
    # may throw an exception

# could have an except block in
the middle

finally:
    # block of code
    # this will always be
executed
```



`try-except-finally` blocks



Example

Example with `try-except-finally`

```
try:
    x= 10
    y= 0
    z= x/y
    print("Answer:", z)
except (ZeroDivisionError):
    print("Can't divide
value with zero.")
finally :
    print("Inside finally
block")
```

Output Exception

```
Can't divide value with zero.
Inside finally block
```

In Python, if an action needs to execute regardless if an error is thrown in a program or not, then we can use a `finally` statement with `try` and `except` statements.

Raising exceptions

raise

Digital Career Institute

Sometimes, an exception has to be thrown in code when a developer wants to write a program that would throw errors if it does not fulfill certain logic.

Python provides **`raise`** statement to **throw** exceptions in code.

The single arguments in the **`raise`** statement shows the exception that has to be raised.

We can raise exceptions in cases such as receiving wrong data or a validation failure.

Steps to `raise` exceptions

- Create an exception of the appropriate type.
- Use the existing built-in exceptions or create your own exception according to our requirement.
- Pass the appropriate data while raising an exception. Execute a raise statement, by providing the `Exception` class.

The syntax for `raise statements:`

```
raise  SomeExceptionClass(<value>)
```

Example

Let's look at a function that will throw an exception if the interest rate is greater than 100.

```
def simple_interest(amount, year, rate):
    if rate > 100:
        raise ValueError(
            f"Your rate is out of range {rate}."
        )
    else:
        interest = (amount * year * rate)/100
        print("Simple interest:", interest)
        return interest

simple_interest(800, 6, 800)
```

Output

```
ValueError: Your interest rate is out of range 800.
```

Exception classes: Custom exceptions

Exception Classes

Digital Career Institute

Sometimes we have to define and `raise` exceptions that are not provided natively by Python.

Such type of exceptions are called **user-defined exception** or **customized exception.**

The user can define custom exceptions by creating a new class. This new exception class has to derive either directly or indirectly from the built-in class `Exception`.

Example

exceptions.py

```
class UnderAgeError(Exception):
    """Raised when the age is below 18."""
    pass
```

The custom exception is defined by simply extending the `Exception` class.

payments.py

```
from exceptions import UnderAgeError

def pay(user, amount):
    if user.age < 18:
        raise UnderAgeError(
            "You must be 18 or older to make"
            " payments."
        )
    else:
        subtract_amount(user, amount)
```

Then, the custom exception can be raised whenever it is needed.

Example

transactions.py

```
from exceptions import UnderAgeError
from payments import pay

def buy(user, item):
    try:
        pay(user, item.price)
        ship(item, user.address)
    except UnderAgeError as error:
        log(error)
        user.call_parents()
        redirect_to("forbidden.html")
```

The custom exception can be catched by importing it and using it in the **`except`** clause.

Because the **`pay`** function raises an error when the user is younger than 18, no transaction will be able to charge any amount to an underage user.

Customizing Exception Classes

Digital Career Institute

```
from exceptions import UnderAgeError
from payments import pay

def buy(user, item):
    try:
        pay(user, item.price)
        ship(item, user.address)
    except UnderAgeError as error:
        log(error)
        user.call_parents()
        redirect_to("forbidden.html")
```

We can **customize** the **classes** by accepting arguments as per our requirements.

Any custom exception class must be extending from **`BaseException`** class or subclass of **`BaseException`**.

The `BaseException` class is, as the name suggests, the base class for all built-in exceptions in Python.

Example

```
class NegativeAgeError(Exception):

    def __init__(self, age):
        self.message = f"Age should not be
negative. Age was {age}."
        super().__init__(self.message)


age = int(input("Enter age: "))
if age < 0:
    raise NegativeAgeError(age)
```

Output

```
Enter age: -28
 Traceback (most recent call last):
   File "/exception.py", line 11, in
     raise NegativeAgeError(age)
   __main__.NegativeAgeError: Age
  should not be negative. Age was -5.
```

At the core of the lesson

- Errors and exceptions allow programmers to debug their code
- Exceptions are logical errors, that can be thrown and caught using `try-except` blocks
- Code that must execute can be put in a `finally` block
- `Raise` statements allow developers to throw errors in their code, this could be in-built python exceptions or custom exceptions
- Developers can create their exceptions classes from the BaseException class to create custom exceptions

Built-in exceptions

Built-in exceptions

Python **automatically** generates many exceptions and errors with bulitin exceptions under the `BaseException` class.

Runtime exceptions, are generally a result of programming errors, such as:

- Reading a file that is not present
- Trying to read data outside the available index of a list
- Dividing an integer value by zero

Please have a look at the list of built-in exception classes on Python Docs:

https://docs.python.org/3/library/exceptions.html#bltin-exceptions

Important Built-in exceptions

AssertionError	Raised when an `assert` statement fails
AttributeError	Raised when attribute assignment or reference fails
EOFError	Raised when input() function hits the end-of-file condition
FloatingPointError	Raised when a floating-point operation fails
GeneratorExit	Raise when a generator's close() method is called
ImportError	Raised when the imported module is not found
IndexError	Raised when the index of a sequence is out of range
KeyError	Raised when a key is not found in a dictionary
KeyboardInterrupt	Raised when the user hits the interrupt key (Ctrl+C or delete)

Important Built-in Exceptions

MemoryError	Raised when an operation runs out of memory
NameError	Raised when a variable is not found in the local or global scope
OSError	Raised when system operation causes system related error
ReferenceError	Raised when a weak reference proxy is used to access a garbage collected referent.

Exception class hierarchy

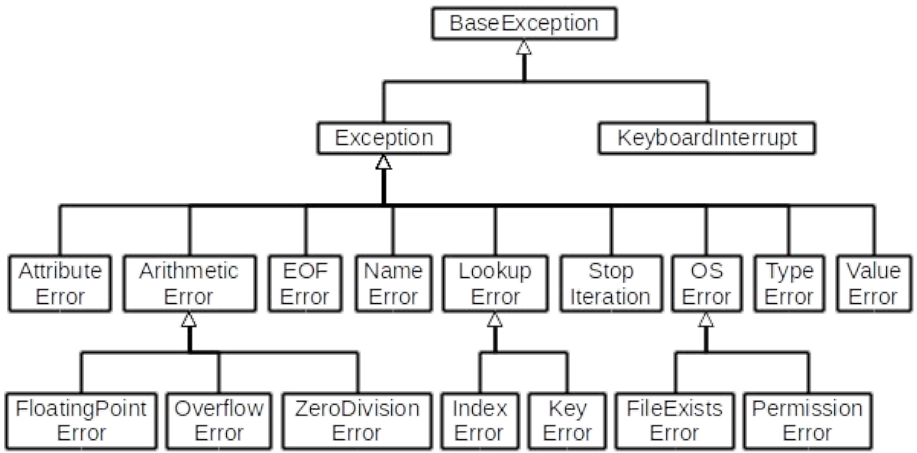
Digital Career Institute

Python has **arranged** the built-in exception into a **class hierarchy** using **inheritance**.

Therefore, any class of exceptions used in an exception statement will also catch errors from its corresponding sub-class as well.

Check out the full list of exception classes and subclasses :
https://docs.python.org/3/library/exceptions.html#exception-hierarchy

Exception class hierarchy - visualisation



Source: [exception_hierarchy.png]

For Example, raising an exception of type LookUp Error will also raise errors of types:

- Index error
- Key error

You can check the class hierarchy of an exception in Python by using the following syntax:

```
<ExceptionClass>.mro()
```

This will return a list of the entire class tree. Try it out!

Example

Digital Career Institute

Both the `lookups()` functions below should give you identical outputs since `IndexError` and `KeyError` are sub-classes of `LookupError`. Try it!

```
def lookups():
    s = [1,4,6]
    try:
        item = s[5]
    except IndexError:
        print("Handled IndexError")
    d = dict(a=1, b=2)
    try:
        value = c["x"]
    except KeyError:
        print("Handled KeyError")
lookups()
```

```
def lookups():
    s = [1,4,6]
    try:
        item = [5]
    except LookupError:
        print("Handled IndexError")
    d = dict(a=1, b=2)
    try:
        value = c["x"]
    except LookupError:
        print("Handled KeyError")
lookups()
```

At the core of the lesson

- There many in-built python exceptions, handling many different types of errors
- These built-in exceptions have sub-classes of “children” exceptions.
- The sub-classes of exceptions can be substituted with the parent class to have the exact same exceptions in a program

Documentation

Documentation

Digital Career Institute

1. Python.org documentation
2. W3Schools

Contact Details
DCI Digital Career Institute gGmbH

Digital Career Institute