Programming Errors

1. **Compilation** (High-level ---> Machine code)

   Syntax Errors: Violates syntax rules.

   Syntax Error

   Error from code not following language rules.

   - **Caused by** missing parenthesis, Incorrect indentation, Misspelled keyword
   - **Detected when** Interpreter/Compiler raises syntax error; code can't run until fixed.
   - **Resolution** is to identify and correct grammatical issues to match syntax rules.
2. **Execution** (Runtime Errors)

   Exceptions: Issues during runtime.

In [1]:
```
# Syntax Error Examples

print 'hello world'
```

```
  Cell In[1], line 3
    print 'hello world'
    ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print
(...)?
```

Syntax Errors: Key Points

- **Missing Symbols**: E.g., colons, brackets ---> disrupts code structure.
- **Misspelled Keywords**: Reserved words ---> unrecognized commands.
- **Incorrect Indentation**: Critical in Python ---> wrong block interpretation.
- **Empty Control Structures**: `if/else` , loops, functions ---> must include executable code or placeholders.

In [2]:
```
a = 5

if a == 3
  print('hello')
```

```
  Cell In[2], line 3
    if a == 3
             ^
SyntaxError: expected ':'
```

In [3]:
```
a = 5

iff a == 3:
  print('hello')
```

```
  Cell In[3], line 3
    iff a == 3:
        ^
SyntaxError: invalid syntax
```

In [4]:
```
a = 5

if a == 3:
print('hello')
```

```
  Cell In[4], line 4
    print('hello')
    ^
IndentationError: expected an indented block after 'if' statement on line
3
```

In [5]:
```
# 1. IndexError: Accessing invalid index.

L = [1, 2, 3]
L[100]
```

```
---------------------------------------------------------------------------
-
IndexError                                Traceback (most recent call las
t)
Cell In[5], line 4
      1 # 1. IndexError: Accessing invalid index.
      3 L = [1, 2, 3]
----> 4 L[100]

IndexError: list index out of range
```

In [6]:
```
# 2. ModuleNotFoundError: Module not found.

import mathi
math.floor(5.3)
```

```
---------------------------------------------------------------------------
-
ModuleNotFoundError                       Traceback (most recent call las
t)
Cell In[6], line 3
      1 # 2. ModuleNotFoundError: Module not found.
----> 3 import mathi
      4 math.floor(5.3)

ModuleNotFoundError: No module named 'mathi'
```

In [7]:
```
# 3. KeyError: Dictionary key not found.

d = {'name':'nitish'}
d['age']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[7], line 4
      1 # 3. KeyError: Dictionary key not found.
      3 d = {'name':'nitish'}
----> 4 d['age']

KeyError: 'age'
```

In [8]:
```
# 4. TypeError: Inappropriate type for operation.

1 + 'a'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[8], line 3
      1 # 4. TypeError: Inappropriate type for operation.
----> 3 1 + 'a'

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

In [9]:
```
# 5. ValueError: Correct type, wrong value.

int('a')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[9], line 3
      1 # 5. ValueError: Correct type, wrong value.
----> 3 int('a')

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

In [10]:
```
# 6. NameError: Undefined name.

print(k)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[10], line 3
      1 # 6. NameError: Undefined name.
----> 3 print(k)

NameError: name 'k' is not defined
```

In [11]:
```
# 7. AttributeError: Invalid attribute reference.

L = [1, 2, 3]
L.upper()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[11], line 4
      1 # 7. AttributeError: Invalid attribute reference.
      3 L = [1, 2, 3]
----> 4 L.upper()

AttributeError: 'list' object has no attribute 'upper'
```

Stacktrace Overview:

- Detailed error info during execution. Includes error type, message, code location (line/file).
- **Purpose** is to help identify and fix issues. Crucial for debugging in development/testing.

Production Considerations:

- **UX:** Avoid displaying to users. Technical jargon can confuse and frustrate.
- **Security:** Exposing can leak sensitive info. Risk of exploitation.

Best Practices:

- Gracefully handle errors. Show user-friendly messages.
- Use stacktraces internally for debugging only.

Exceptions in Programming:

Runtime issues disrupting execution. Require immediate handling for stability.

Common Issues:

- *Memory overflow*: Exceeds memory limits.

- *Division by zero*: Undefined operations.
- *Database errors*: Connection/query failures.

Importance: Prevents crashes, ensures stability, and improves reliability.

Python Handling

In [37]:
```
# Create file and write text

with open('sample.txt', 'w') as f:
  f.write('hello world')
```

In [3]:
```
# Try-Catch Demo

try:
    with open('sample1.txt', 'r') as f:
        print(f.read())
except:
    print('sorry file not found')
```

```
sorry file not found
```

`try-except` Blocks

Purpose:

- *Mitigate Risks*: File perms, network issues.
- *Enhance Robustness*: Avoid crashes, manage errors.
- *Graceful Handling*: Recover from issues.

Structure:

- **`try` Block**: Risky ops (file I/O, DB).
- **`except` Block**: Catches exceptions from `try`.

Benefits:

- *Reliability*: Avoid crashes.
- *Clean Code*: Error-handling separate.
- *Resilience*: Recover from errors.

Best Practices:

- Specific exceptions (`FileNotFoundError`, `ConnectionError`).
- Appropriate messages or fallbacks in `except`.

In [24]:
```python
# Catching Specific Exceptions ---> informing users about errors, improving

try:
    m = 5
    f = open('sample1.txt', 'r')
    print(f.read())
    print(k)
    print(5 / 0)
    L = [1, 2, 3]
    L[100]
except FileNotFoundError:
    print('file not found')
except NameError:
    print('variable not defined')
except ZeroDivisionError:
    print("can't divide by 0")
except Exception as e:
    print(e)
```

```
file not found
```

In [38]:
```python
# `else` in Try-Except

try:
    f = open('sample.txt', 'r')
except FileNotFoundError:
    print('file nai mili')
except Exception:
    print('kuch to lafda hai')
else:
    print(f.read())
```

```
hello world
```

`try`, `else`, and `finally` Blocks

`try` Block: Executes risky code; avoids abrupt halts on errors.

`else` Block: Runs if `try` succeeds; executes only when no exceptions.

`except` Block: Manages errors from `try` .

`finally` Block: Executes regardless of exceptions; ensures cleanup (e.g., close files, release resources).

`try` , `else` , `finally` = Structured exception handling.

In [42]:
```
# `finally`

try:
  f = open('sample1.txt', 'r')
except FileNotFoundError:
  print('file nai mili')
except Exception:
  print('kuch to lafda hai')
else:
  print(f.read())
finally:
  print('ye to print hoga hi')
```

```
file nai mili
ye to print hoga hi
```

`raise` Keyword

Trigger exceptions manually.

Custom Exceptions: Pass values for context.

Error Control: Enhance robust design and manage unexpected issues.

In [43]:
```
raise ZeroDivisionError('aise hi try kar raha hu')

# Java Equivalents:

# `try`    ---> `try`
# `except` ---> `catch`
# `raise`  ---> `throw`
```

```
---------------------------------------------------------------------------
-
ZeroDivisionError                         Traceback (most recent call las
t)
Cell In[43], line 1
----> 1 raise ZeroDivisionError('aise hi try kar raha hu')

ZeroDivisionError: aise hi try kar raha hu
```

In [49]:
```
class Bank:

  def __init__(self, balance):
    self.balance = balance

  def withdraw(self, amount):
    if amount < 0:
      raise Exception('amount cannot be -ve')
    if self.balance < amount:
      raise Exception('paise nai hai tere paas')
    self.balance = self.balance - amount

obj = Bank(10000)
try:
  obj.withdraw(15000)
except Exception as e:
  print(e)
else:
  print(obj.balance)
```

```
paise nai hai tere paas
```

In [ ]:
```
`raise`  ---> Trigger exceptions.
`except` ---> Handle exceptions.

Enhances app robustness by managing errors, preventing abrupt terminations.
```

exception hierarchy in python

In [ ]:
```
               BaseException
                     |
                     |
           +---------+---------+
           |                   |
           |                   |
       Exception      KeyboardInterrupt
           |                                    +--------- FloatingPointError
           |                                    |
           +-------- ArithmeticError -----------+--------- OverflowError
           |                                    |
           |                                    +--------- ZeroDivisionError
           |
           |                                    +--------- IndexError
           +---------- LookupError -------------|
           |                                    +--------- KeyError
           |
           |                                    +--------- FileExists Error
           +------------ OSError ---------------|
                                                +--------- Permission Error
```

Python allows creating custom exceptions, which means you can define your own types of errors.

In [53]:
```python
class MyException(Exception):
  def __init__(self, message):
    print(message)

class Bank:
  def __init__(self, balance):
    self.balance = balance

  def withdraw(self, amount):
    if amount < 0:
      raise MyException('amount cannot be -ve')
    if self.balance < amount:
      raise MyException('paise nai hai tere paas')
    self.balance = self.balance - amount

obj = Bank(10000)
try:
  obj.withdraw(15000)
except MyException as e:
  pass
else:
  print(obj.balance)
```

```
paise nai hai tere paas
```

Custom Classes: Why & Benefits

Purpose:

- Full control over app structure & behavior
- Ideal for custom login/registration systems

Benefits:

1. **Security**: Custom security measures, e.g., device signature management, auto log-out on unrecognized devices
2. **Functionality**: Tailored features, e.g., user input management (name, email, password), device signature handling

Implementation:

- Control over security protocols
- Enables advanced security features

simple example

In [3]:
```
class SecurityError(Exception):

  def __init__(self, message):
    print(message)

  def logout(self):
    print('logout')

class Google:

  def __init__(self, name, email, password, device):
    self.name = name
    self.email = email
    self.password = password
    self.device = device

  def login(self, email, password, device):
    if device != self.device:
      raise SecurityError('bhai teri to lag gayi')
    if email == self.email and password == self.password:
      print('welcome')
    else:
      print('login error')

obj = Google('nitish', 'nitish@gmail.com', '1234', 'android')

try:
  obj.login('nitish@gmail.com', '1234', 'windows')
except SecurityError as e:
  e.logout()
else:
  print(obj.name)
finally:
  print('database connection closed')
```

```
bhai teri to lag gayi
logout
database connection closed
```

In [ ]:
```
```