migo_8_files_exceptions_builtins

July 6, 2024

Exceptions

events detected during program execution that disrupt the normal flow of instructions.

[ ]:
```
Raising Exceptions
```

[1]:
```
a=int(input("Please enter positive number"))
if a <0:
    raise ValueError("Invalid input,negative number")
print(a)
```

```
Please enter positive number -3
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[1], line 3
      1 a=int(input("Please enter positive number"))
      2 if a <0:
----> 3     raise ValueError("Invalid input,negative number")
      4 print(a)

ValueError: Invalid input,negative number
```

[2]:
```
x="lala"
if not type(x) is int:
    raise TypeError("Only integers are allowed")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[2], line 3
      1 x="lala"
      2 if not type(x) is int:
----> 3     raise TypeError("Only integers are allowed")

TypeError: Only integers are allowed
```

Handling Exceptions

[ ]:
```
Use try, except, else, and finally blocks to handle exceptions
```

[3]:
```
try:
    x = 1 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")

except ValueError:
    print("No integer")

#except(ZeroDivisionError,ValueError):  # multiple exceptions
#    print("divide by zero or no integer)

except:
    print("Some thing wrong")
else:                                   # no exception was raised
    print("No exception occurred")
finally:                                #always and  'cleanup' handler
    print("This block always executes")
```

```
Cannot divide by zero
This block always executes
```

[4]:
```
def my_div(a,b):
    try:
        res = a/b
    except Exception as e:
        print(repr(e))  # e.__repr__()
    else:
        print("done")
        return res
    finally:
        print("end of func") # 'cleanup' handler
print(my_div(3,6))
print(my_div("sd",5))
print(my_div(2,0))
```

```
done
end of func
0.5
TypeError("unsupported operand type(s) for /: 'str' and 'int'")
end of func
None
ZeroDivisionError('division by zero')
end of func
None
```

print exceptions hierachy

[ ]:
```
def print_exception_hierarchy(base_class, indent=0):
    print(' ' * indent + base_class.__name__)
    for subclass in base_class.__subclasses__():
        print_exception_hierarchy(subclass, indent + 4)

print_exception_hierarchy(BaseException)
```

[ ]:
```
Files
```

open ,close read,readline,readlines,write,writelines

[ ]:
```
modes:r,w,a,t,b,+
```

[ ]:
```
txt files
```

[7]:
```
st=open("C:lala.txt","wt") #t default
st.write("Hello world\n")
st.write("lala in lala-land")
st.close()

st=open("C:lala.txt","rt")
print(st.readlines())
st.close()

#use context managers
with open("C:lala.txt") as my_file: # this close the file even if an exception␣
 ↪is raised
    print(my_file.read())
```

```
['Hello world\n', 'lala in lala-land']
Hello world
lala in lala-land
```

Iterating Over File Lines

[8]:
```
with open("C:lala.txt", 'r') as file:
    for line in file:
        print(line.strip())
```

```
Hello world
lala in lala-land
```

methods :seex,flush,truncate(size=None): Resizes the file to a specified size

[ ]:
```
bin files
```

[9]:
```
d = bytearray(b'Hello World')
d.extend([32,97,98,99])
d.append(0x64)
with open('example.bin', 'wb') as file:
```

```
    file.write(d)
    file.write(b"\nlala")
read_bit_data = bytearray()
with open('example.bin', 'rb') as file:
    read_byte_data = bytearray(file.read())
print(read_byte_data)
```

```
bytearray(b'Hello World abcd\nlala')
```

anonymous function lamda

[10]:
```
f1=lambda a:a*a
f2=lambda a,b:a*b
print(f1(2),f2(12,3))
suc=lambda s:s.strip().upper()
print(suc("  hello world!  "))
```

```
4 36
HELLO WORLD!
```

lambda if-else

[11]:
```
result = lambda x : f"{x} is even" if x %2==0 else f"{x} is odd"
print(result(12),result(11))
par=lambda x:"zero" if x==0  else ( "even"  if x%2==0 else "odd")
print(par(12),par(11),par(0))
```

```
12 is even 11 is odd
even odd zero
```

return lambda

[12]:
```
def myfunc(n):
            return lambda a : a * n
mydoubler= myfunc(2)
myttriler= myfunc(3)
print(mydoubler(11),myttriler(11))
```

```
22 33
```

iterators

iter(),**next**():must raise the StopIteration exception if nomore elements

[13]:
```
L = [1, 2, 3]
it = iter(L)
print(it)
print(it.__next__(),next(it),next(it))
next(it)
```

```
<list_iterator object at 0x000002812CB48BB0>
1 2 3
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[13], line 5
      3 print(it)
      4 print(it.__next__(),next(it),next(it))
----> 5 next(it)

StopIteration:
```

[14]:
```
l=[1,2,3,4]
it=iter(l)
a,b,c,d=iter(l)
print(a,b,c,d)
t=tuple(it)
print(t)
for item in iter(l):
    print(item,end=" ")
```

```
1 2 3 4
(1, 2, 3, 4)
1 2 3 4
```

Generators: simplify the task of writing iterators

yeild

[15]:
```
def generate_ints(N):
    for i in range(N):
        yield i
gen = generate_ints(3)
print(gen)
print(gen.__next__(),next(gen),next(gen))
print(next(gen))
```

```
<generator object generate_ints at 0x000002812D1372A0>
0 1 2
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[15], line 7
      5 print(gen)
      6 print(gen.__next__(),next(gen),next(gen))
----> 7 print(next(gen))

StopIteration:
```

send allows you to send a value back into the generator to affect its behavior or state.

[16]:
```python
def simple_generator():
    while True:
        value = (yield)
        if value is None:
            break
        print(f"Received value: {value}")

gen = simple_generator()
next(gen)  # Prime the generator (start it)
gen.send(10)  # Output: Received value: 10
gen.send(20) ; gen.send(30)
gen.send(None)  # Stop the generator
```

```
Received value: 10
Received value: 20
Received value: 30
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[16], line 12
     10 gen.send(10)  # Output: Received value: 10
     11 gen.send(20) ; gen.send(30)
---> 12 gen.send(None)  # Stop the generator

StopIteration:
```

[17]:
```python
def accumulator():
    total = 0
    while True:
        increment = yield total
        if increment is None:
            break
        total += increment

gen = accumulator()
print(next(gen))

print(gen.send(5))
print(gen.send(10))
print(gen.send(3))
gen.send(None)
```

```
0
5
15
18
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[17], line 15
     13 print(gen.send(10))  # Output: 15 (total after adding 10)
     14 print(gen.send(3))  # Output: 18 (total after adding 3)
---> 15 gen.send(None)  # Stop the generator

StopIteration:
```

[18]:
```python
def accumulator(initial_increment=0):
    total = initial_increment
    while True:
        increment = yield total
        if increment is not None:
            total += increment
        else:
            total += 1

gen = accumulator()
print(next(gen))
print(next(gen))
print(gen.send(50))  # Output: 52 (Incremented by 50)
print(next(gen))
print(gen.send(10))
print(next(gen))
```

```
0
1
51
52
62
63
```

builtins functions

dir: provide an overview of the available namespace. List the names of the attributes and methods of an object.

[ ]:
```python
print(dir())
dir(__builtins__)
```

some of them

sorted: return a new sorted list from the elements of any iterable (such as lists, tuples, or strings).

[7]:
```python
# sorted(iterable, key=None, reverse=False)

s="sdfgfgtrewfvc"
```

```
print(sorted(s))
s=("1234","1","12","123","la","aaa","ba")
print(sorted(s,key=len,reverse=True))
```

```
['c', 'd', 'e', 'f', 'f', 'f', 'g', 'g', 'r', 's', 't', 'v', 'w']
['1234', '123', 'aaa', '12', 'la', 'ba', '1']
```

exec: execute a string containing Python code (very dangerous)

[6]:
```
st="""
x=3
y=5
print(f"The sum of {x},{y} is {x+y}")
"""
exec(st)
```

```
The sum of 3,5 is 8
```

map: apply a given function to all items in an iterable and return a map object (which is an iterator)

[5]:
```
# map(function, iterable, ...)

numbers = [1, 2, 3, 4, 5]
squared_numbers = map(lambda x: x * x, numbers)
print(list(squared_numbers))

numbers1 = [1, 2, 3,11,12] ; numbers2 = [4, 5, 6]
summed_numbers = map(lambda x, y: x + y, numbers1, numbers2)
print(list(summed_numbers))
```

```
[1, 4, 9, 16, 25]
[5, 7, 9]
```

filter: used to construct an iterator from elements of an iterable for which a function returns true.

[4]:
```
# filter(function, iterable)

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers))

words = ["apple", "banana", "cherry", "date"]
words_with_a = filter(lambda word: 'a' in word, words)
print(list(words_with_a))
```

```
[2, 4, 6, 8, 10]
['apple', 'banana', 'date']
```

zip: used to combine multiple iterables (such as lists, tuples, etc.) into a single iterator of tuples

[3]:
```python
# zip(iterable1, iterable2, ...)
list1 = [1, 2, 3, 4];list2 = ['a', 'b']
zipped = zip(list1, list2)
print(list(zipped))  # Output: [(1, 'a'), (2, 'b')]

list1 = [1, 2, 3];list2 = ['a', 'b', 'c'];list3 = [0.1, 0.2, 0.3]
zipped = zip(list1, list2, list3)
print(list(zipped))  # Output: [(1, 'a', 0.1), (2, 'b', 0.2), (3, 'c', 0.3)]

#using loop
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']
for number, letter in zip(list1, list2):
    print(f"Number: {number}, Letter: {letter}")
```

```
[(1, 'a'), (2, 'b')]
[(1, 'a', 0.1), (2, 'b', 0.2), (3, 'c', 0.3)]
Number: 1, Letter: a
Number: 2, Letter: b
Number: 3, Letter: c
```

Unzipping: Separating a list of tuples back into individual lists using zip(*zipped)

[2]:
```python
# zip(*zipped)

zipped = [(1, 'a'), (2, 'b'), (3, 'c')]
list1, list2 = zip(*zipped)
print(list1)
print(list2)
```

```
(1, 2, 3)
('a', 'b', 'c')
```

enumerate: adds a counter to an iterable and returns it as an enumerate object.

[1]:
```python
# enumerate(iterable, start=0)

items = ['apple', 'banana', 'cherry']
for index, value in enumerate(items, start=1):
    print(index, value)

items = ['apple', 'banana', 'cherry']
indexed_items = list(enumerate(items))
print(indexed_items)
```

```
1 apple
2 banana
3 cherry
[(0, 'apple'), (1, 'banana'), (2, 'cherry')]
```

[ ]: