Namespaces and Exceptions, revisited

- Introduction to Database Access
- Encapsulation in Functions
- Global versus Local Namespaces
- Exceptional Control Flow

SQLITE 3 FOR DATABASE ACCESS

Standard Library module `sqlite3`

The Python Standard Library includes module `sqlite3` that provides an API for accessing database files

- It is an interface to a library of functions that accesses the database files directly

```
>>> import sqlite3
>>> con = sqlite3.connect('web.db')
```

sqlite3 function `connect()` takes as input the name of a database and returns an object of type `Connection`, a type defined in module `sqlite3`

- The `Connection` object `con` is associated with database file `web.db`
- If database file `web.db` does not exists in the current working directory, a new database file `web.db` is created

Standard Library module `sqlite3`

The Python Standard Library includes module `sqlite3` that provides an API for accessing database files

- It is an interface to a library of functions that accesses the database files directly

```
>>> import sqlite3
>>> con = sqlite3.connect('web.db')
>>> cur = con.cursor()
```

`Connection` method `cursor()` returns an object of type `Cursor`, another type defined in the module `sqlite3`

- `Cursor` objects are responsible for executing SQL statements

Standard Library module `sqlite3`

The Python Standard Library includes module `sqlite3` provides an API for accessing database files

- It is an interface to a library of functions that accesses the database files directly

```
>>> import sqlite3
>>> con = sqlite3.connect('web.db')
>>> cur = con.cursor()
>>> cur.execute("CREATE TABLE Keywords (Url text, Word text, Freq int)")
<sqlite3.Cursor object at 0x100575730>
>>> cur.execute("INSERT INTO Keywords VALUES ('one.html', 'Beijing', 3)")
<sqlite3.Cursor object at 0x100575730>
```

The `Cursor` class supports method `execute()` which takes an SQL statement as a string, and executes it

Hardcoded values

Parameter substitution

In general, the values used in an SQL statement will not be hardcoded in the program but come from Python variables

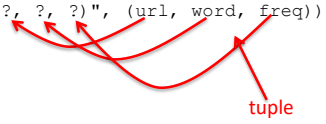
```
>>> cur.execute("INSERT INTO Keywords VALUES ('one.html', 'Beijing', 3)")
<sqlite3.Cursor object at 0x100575730>
>>> url, word, freq = 'one.html', 'Paris', 5
>>>
```

Parameter substitution

Parameter substitution is the technique used to construct SQL statements that make use of Python variable values

- similar to string formatting

```
>>> cur.execute("INSERT INTO Keywords VALUES ('one.html', 'Beijing', 3)")
<sqlite3.Cursor object at 0x100575730>
>>> url, word, freq = 'one.html', 'Paris', 5
>>> cur.execute("INSERT INTO Keywords VALUES (?, ?, ?)", (url, word, freq))
<sqlite3.Cursor object at 0x100575730>
```



Parameter substitution

Parameter substitution is the technique used to construct SQL statements that make use of Python variable values

- similar to string formatting

```
>>> cur.execute("INSERT INTO Keywords VALUES ('one.html', 'Beijing', 3)")
<sqlite3.Cursor object at 0x100575730>
>>> url, word, freq = 'one.html', 'Paris', 5
>>> cur.execute("INSERT INTO Keywords VALUES (?, ?, ?)", (url, word, freq))
<sqlite3.Cursor object at 0x100575730>
>>> record = ('one.html','Chicago', 5)
>>> cur.execute("INSERT INTO Keywords VALUES (?, ?, ?)", record)
<sqlite3.Cursor object at 0x100575730>
```

Parameter substitution

Changes to a database file are not written to the database file immediately; they are only recorded temporarily, in memory

In order to ensure that the changes are written to the database file, the `commit()` method must be called on the `Connection` object

```
>>> cur.execute("INSERT INTO Keywords VALUES ('one.html', 'Beijing', 3)")
<sqlite3.Cursor object at 0x100575730>
>>> url, word, freq = 'one.html', 'Paris', 5
>>> cur.execute("INSERT INTO Keywords VALUES (?, ?, ?)", (url, word, freq))
<sqlite3.Cursor object at 0x100575730>
>>> record = ('one.html','Chicago', 5)
>>> cur.execute("INSERT INTO Keywords VALUES (?, ?, ?)", record)
<sqlite3.Cursor object at 0x100575730>
>>> con.commit()
>>> con.close()
```

A database file should be closed just like any other file

Querying a database

The result of a query is stored in the `Cursor` object

To obtain the result as a list of `tuple` objects, `Cursor` method `fetchall()` is used

```
>>> import sqlite3
>>> con = sqlite3.connect('links.db')
>>> cur = con.cursor()
>>> cur.execute('SELECT * FROM Keywords')
<sqlite3.Cursor object at 0x102686960>
>>> cur.fetchall()
[('one.html', 'Beijing', 3), ('one.html', 'Paris', 5), ('one.html',
'Chicago', 5), ('two.html', 'Bogota', 5), ('two.html', 'Beijing', 2),
('two.html', 'Paris', 1), ('three.html', 'Chicago', 3), ('three.html',
'Beijing', 6), ('four.html', 'Chicago', 3), ('four.html', 'Paris', 2),
('four.html', 'Nairobi', 5), ('five.html', 'Nairobi', 7), ('five.html',
'Bogota', 2)]
>>>
```

Querying a database

An alternative is to iterate over the `Cursor` object

```
>>> cur.execute('SELECT * FROM Keywords')
<sqlite3.Cursor object at 0x102686960>
>>> for record in cur:
        print(record)


('one.html', 'Beijing', 3)
('one.html', 'Paris', 5)
('one.html', 'Chicago', 5)
('two.html', 'Bogota', 5)
('two.html', 'Beijing', 2)
('two.html', 'Paris', 1)
('three.html', 'Chicago', 3)
('three.html', 'Beijing', 6)
('four.html', 'Chicago', 3)
('four.html', 'Paris', 2)
('four.html', 'Nairobi', 5)
('five.html', 'Nairobi', 7)
('five.html', 'Bogota', 2)
>>>
```


Querying a database

Parameter substitution is again used whenever Python variable values are needed in the SQL statement

```
>>> word = 'Paris'
>>> cur.execute('SELECT Url FROM Keywords WHERE Word = ?', (word,))
<sqlite3.Cursor object at 0x102686960>
>>> cur.fetchall()
[('one.html',), ('two.html',), ('four.html',)]
>>> word, n = 'Beijing', 2
>>> cur.execute("SELECT * FROM Keywords WHERE Word = ? AND Freq > ?", (word, n))
<sqlite3.Cursor object at 0x102686960>
>>> cur.fetchall()
[('one.html', 'Beijing', 3), ('three.html', 'Beijing', 6)]
>>>
```

The purpose of functions

Wrapping code into functions has several desirable goals:

- **Modularity:** The complexity of developing a large program can be dealt with by breaking down the program into smaller, simpler, self-contained pieces. Each smaller piece (e.g., function) can be designed, implemented, tested, and debugged independently.
- **Code reuse:** A fragment of code that is used multiple times in a program—or by multiple programs— should be packaged in a function. The program ends up being shorter, with a single function call replacing a code fragment, and clearer, because the name of the function can be more descriptive of the action being performed by the code fragment. Debugging also becomes easier because a bug in the code fragment will need to be fixed only once.
- **Encapsulation:** A function hides its implementation details from the user of the function; removing the implementation details from the developer's radar makes her job easier.

Encapsulation through local variables

Encapsulation makes modularity and code reuse possible

```
>>> x
Traceback (most recent call last):
  File "<pyshell#62>", line 1, in <module>
    x
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<pyshell#63>", line 1, in <module>
    y
NameError: name 'y' is not defined
>>> res = double(5)
x = 2, y = 5
>>> x
Traceback (most recent call last):
  File "<pyshell#66>", line 1, in <module>
    x
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    y
NameError: name 'y' is not defined
```

Before executing function `double()`, variables `x` and `y` do not exist

```
def double(y):
    x=2
    print('x = {}, y = {}'.format(x,y))
    return x*y
```

After executing function `double()`, variables `x` and `y` still do not exist

`x` and `y` exist only during the execution of function call `double(5)`; they are said to be local variables of function `double()`

Function call namespace

```
>>> x
Traceback (most recent call last):
  File "<pyshell#62>", line 1, in <module>
    x
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<pyshell#63>", line 1, in <module>
    y
NameError: name 'y' is not defined
>>> res = double(5)
x = 2, y = 5
>>> x
Traceback (most recent call last):
  File "<pyshell#66>", line 1, in <module>
    x
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    y
NameError: name 'y' is not defined
```

Even during the execution of `double()`, local variables `x` and `y` are invisible outside of the function!

```
def double(y):
    x=2
    print('x = {}, y = {}'.format(x,y))
    return x*y
```


Function call namespace

```
>>> x, y = 20, 50
>>> res = double(5)
x = 2, y = 5
>>> x, y
(20, 50)
>>>
```

How is it possible that the values of `x` and `y` do not interfere with each other?

Even during the execution of `double()`, local variables `x` and `y` are invisible outside of the function!

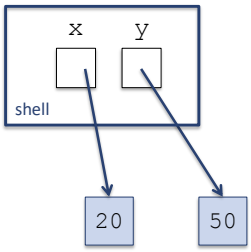
```
def double(y):
    x=2
    print('x = {}, y = {}'.format(x,y))
    return x*y
```

Function call namespace

```
>>> x, y = 20, 50
>>> res = double(5)
```

Even during the execution of `double()`, local variables `x` and `y` are invisible outside of the function!

```
def double(y):
    x=2
    print('x = {}, y = {}'.format(x,y))
    return x*y
```



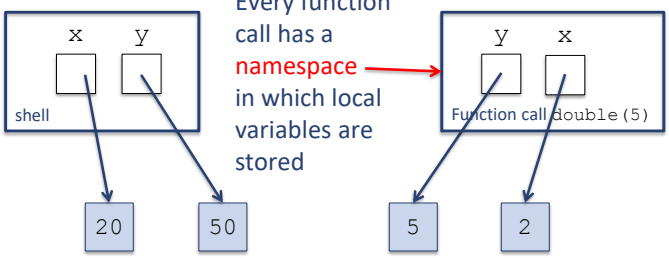
Function call namespace

```
>>> x, y = 20, 50
>>> res = double(5)
x = 2, y = 5
>>> x, y
(20, 50)
>>>
```

Even during the execution of `double()`, local variables `x` and `y` are invisible outside of the function!

```
def double(y):
    x=2
    print('x = {}, y = {}'.format(x,y))
    return x*y
```

Every function call has a namespace in which local variables are stored



Function call namespace

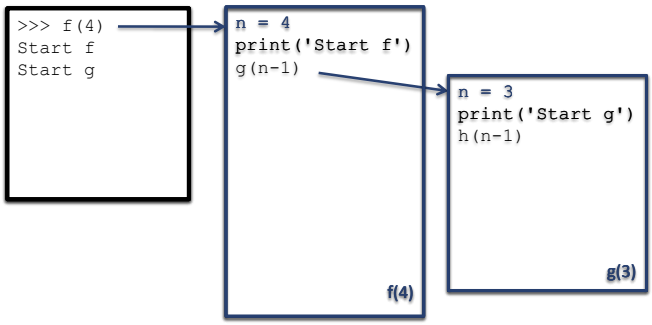
Every function call has a namespace in which local variables are stored

Note that there are several active values of n, one in each namespace; how are all the namespaces managed by Python?

```
def h(n):
    print('Start h')
    print(1/n)
    print(n)

def g(n):
    print('Start g')
    h(n-1)
    print(n)

def f(n):
    print('Start f')
    g(n-1)
    print(n)
```



Function call namespace

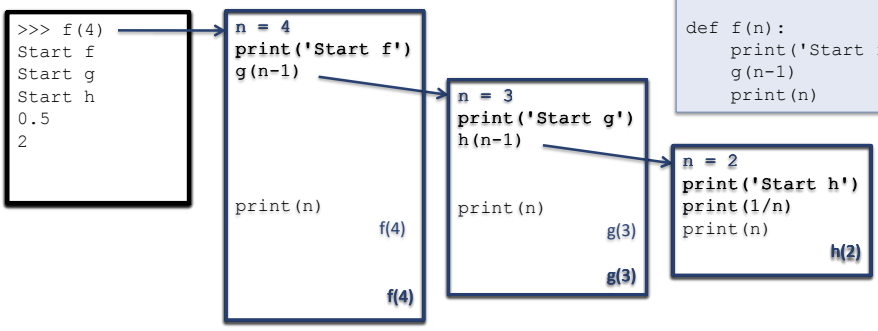
Every function call has a namespace in which local variables are stored

Note that there are several active values of n, one in each namespace; how are all the namespaces managed by Python?

```
def h(n):
    print('Start h')
    print(1/n)
    print(n)

def g(n):
    print('Start g')
    h(n-1)
    print(n)

def f(n):
    print('Start f')
    g(n-1)
    print(n)
```



Function call namespace

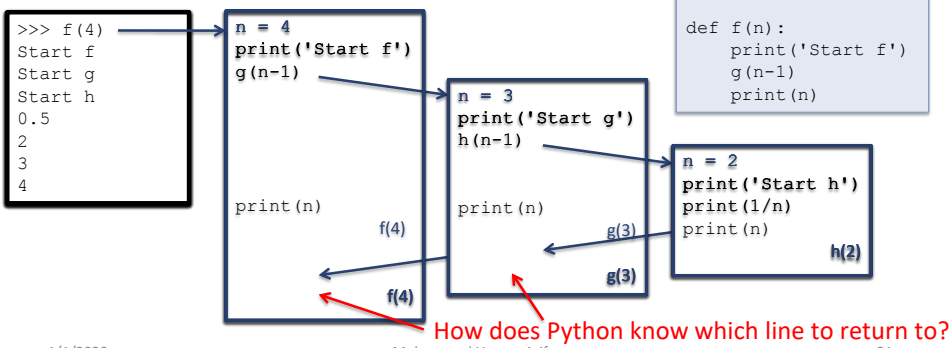
Every function call has a namespace in which local variables are stored

Note that there are several active values of n, one in each namespace; how are all the namespaces managed by Python?

```
def h(n):
    print('Start h')
    print(1/n)
    print(n)

def g(n):
    print('Start g')
    h(n-1)
    print(n)

def f(n):
    print('Start f')
    g(n-1)
    print(n)
```



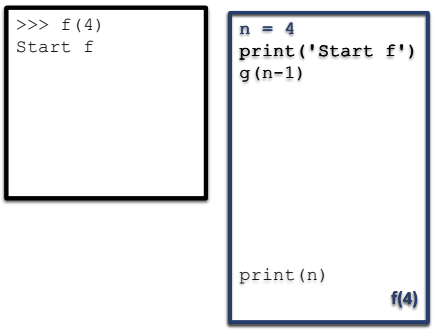
How does Python know which line to return to?

Program stack

The system dedicates a chunk of memory to the program stack; its job is to remember the values defined in a function call and …

```
1. def h(n):
2.     print('Start h')
3.     print(1/n)
4.     print(n)
5.
6. def g(n):
7.     print('Start g')
8.     h(n-1)
9.     print(n)
10.
11. def f(n):
12.     print('Start f')
13.     g(n-1)
14.     print(n)
```

Program stack



Program stack

The system dedicates a chunk of memory to the program stack; its job is to remember the values defined in a function call and …

```
>>> f(4)
Start f
Start g
```

```
n = 4
print('Start f')
g(n-1)

print(n)
```
f(4)

```
n = 3
print('Start g')
h(n-1)

print(n)
```
g(3)

line = 14
n = 4

Program stack

```
1. def h(n):
2.     print('Start h')
3.     print(1/n)
4.     print(n)
5.
6. def g(n):
7.     print('Start g')
8.     h(n-1)
9.     print(n)
10.
11. def f(n):
12.     print('Start f')
13.     g(n-1)
14.     print(n)
```


Program stack

The system dedicates a chunk of memory to the program stack; its job is to remember the values defined in a function call and …

… the statement to be executed after g(n-1) returns

```
>>> f(4)
Start f
Start g
Start h
0.5
2
3
4
```

```
n = 4
print('Start f')
g(n-1)

print(n)
```
f(4)

```
n = 3
print('Start g')
h(n-1)

print(n)
```
g(3)

```
n = 2
print('Start h')
print(1/n)
print(n)
```
h(2)

line = 9
n = 3
line = 14
n = 4

Program stack

```
1. def h(n):
2.     print('Start h')
3.     print(1/n)
4.     print(n)
5.
6. def g(n):
7.     print('Start g')
8.     h(n-1)
9.     print(n)
10.
11. def f(n):
12.     print('Start f')
13.     g(n-1)
14.     print(n)
```

Scope and global vs. local namespace

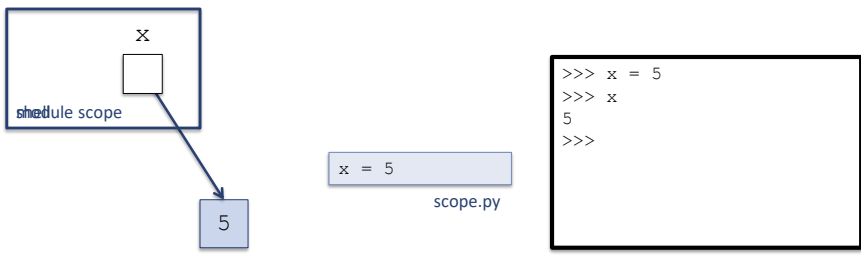
Every function call has a namespace associated with it.

- This namespace is where names defined during the execution of the function (e.g., local variables) live.
- The scope of these names (i.e., the space where they live) is the namespace of the function.

In fact, every name in a Python program has a scope

- Whether the name is of a variable, function, class, …
- Outside of its scope, the name does not exist, and any reference to it will result in an error.
- Names assigned/defined in the interpreter shell or in a module and outside of any function are said to have global scope.

Scope and global vs. local namespace



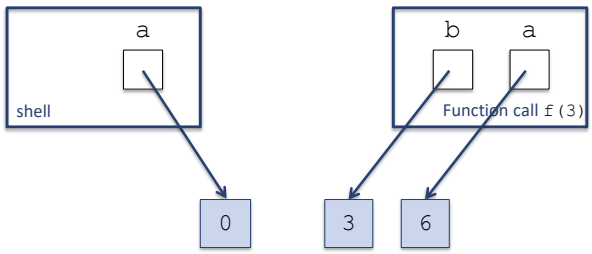
In fact, every name in a Python program has a scope

- Whether the name is of a variable, function, class, …
- Outside of its scope, the name does not exist, and any reference to it will result in an error.
- Names assigned/defined in the interpreter shell or in a module and outside of any function are said to have global scope. Their scope is the namespace associated with the shell or the whole module. Variables with global scope are referred to as global variables.

Example: variable with local scope

```
def f(b):        # f has global scope, b has local scope
    a = 6        # this a has scope local to function call f()
    return a*b   # this a is the local a

a = 0            # this a has global scope
print('f(3) = {}'.format(f(3)))
print('a is {}'.format(a))        # global a is still 0
```

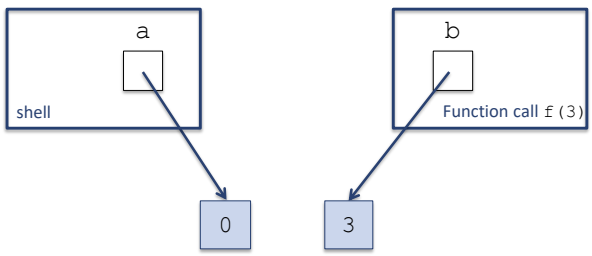



```
>>> === RESTART ====
>>>
f(3) = 18
a is 0
>>>
```


Example: variable with global scope

```
def f(b):        # f has global scope, b has local scope
    return a*b   # this a is the global a

a = 0            # this a has global scope
print('f(3) = {}'.format(f(3)))
print('a is {}'.format(a))        # global a is still 0
```




```
>>> === RESTART ====
>>>
f(3) = 0
a is 0
>>>
```

How Python evaluates names

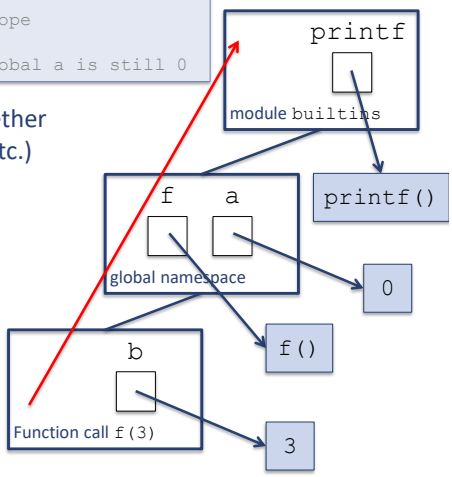
```
def f(b):       # f has global scope, b has local scope
    return a*b  # this a is the global a

a = 0           # this a has global scope
print('f(3) = {}'.format(f(3)))
print('a is {}'.format(a))        # global a is still 0
```

How does the Python interpreter decide whether to evaluate a name (of a variable, function, etc.) as a local or as a global name?

Whenever the Python interpreter needs to evaluate a name, it searches for the name definition in this order:

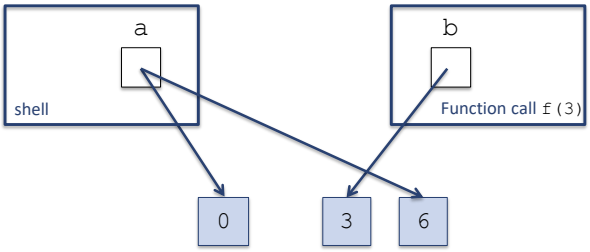
1. First the enclosing function call namespace
2. Then the global (module) namespace
3. Finally the namespace of module `builtins`



Modifying a global variable inside a function

```
def f(b):
    global a    # all references to a in f() are to the global a
    a = 6       # global a is changed
    return a*b  # this a is the global a

a = 0           # this a has global scope
print('f(3) = {}'.format(f(3)))
print('a is {}'.format(a))        # global a has been changed to 6
```




```
>>> === RESTART ====
>>>
f(3) = 18
a is 6
>>>
```

Exceptions, revisited

Recall that when the program execution gets into an erroneous state, an exception object is created

- This object has a type that is related to the *type* of error
- The object contains information about the error
- The default behavior is to print this information and interrupt the execution of the statement that "caused" the error

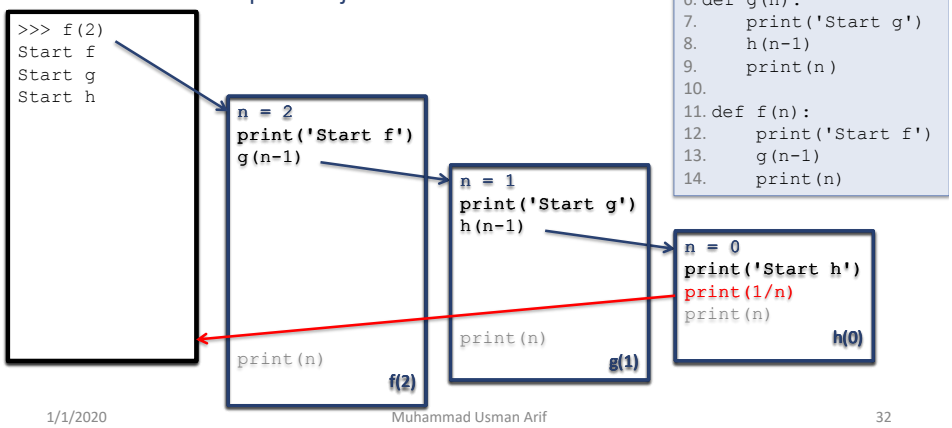
The reason behind the term "exception" is that when an error occurs and an exception object is created, the normal execution flow of the program is interrupted and execution switches to the exceptional control flow

Exceptional control flow

Exceptional control flow

The default behavior is to interrupt the execution of each "active" statement and print the error information contained in the exception object.

```
1. def h(n):
2.     print('Start h')
3.     print(1/n)
4.     print(n)
5.
6. def g(n):
7.     print('Start g')
8.     h(n-1)
9.     print(n)
10.
11. def f(n):
12.     print('Start f')
13.     g(n-1)
14.     print(n)
```

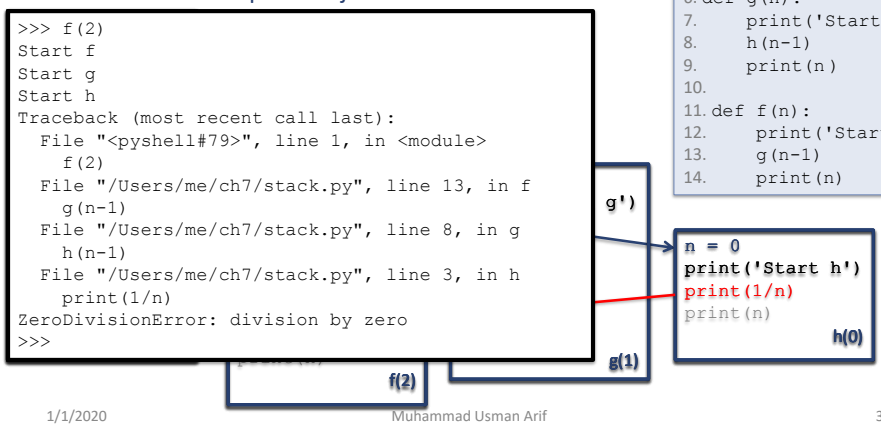


Exceptional control flow

The default behavior is to interrupt the execution of each "active" statement and print the error information contained in the exception object.

```
>>> f(2)
Start f
Start g
Start h
Traceback (most recent call last):
  File "<pyshell#79>", line 1, in <module>
    f(2)
  File "/Users/me/ch7/stack.py", line 13, in f
    g(n-1)
  File "/Users/me/ch7/stack.py", line 8, in g
    h(n-1)
  File "/Users/me/ch7/stack.py", line 3, in h
    print(1/n)
ZeroDivisionError: division by zero
>>>
```

```
1. def h(n):
2.     print('Start h')
3.     print(1/n)
4.     print(n)
5.
6. def g(n):
7.     print('Start g')
8.     h(n-1)
9.     print(n)
10.
11. def f(n):
12.     print('Start f')
13.     g(n-1)
14.     print(n)
```



Catching and handling exceptions

It is possible to override the default behavior (print error information and "crash") when an exception is raised, using `try/except` statements

```
strAge = input('Enter your age: ')
intAge = int(strAge)
print('You are {} years old.'.format(intAge))
```

Default behavior:

```
>>> ========================= RESTART ========================
>>>
Enter your age: fifteen
Traceback (most recent call last):
  File "/Users/me/age1.py", line 2, in <module>
    intAge = int(strAge)
ValueError: invalid literal for int() with base 10: 'fifteen'
>>>
```

Catching and handling exceptions

It is possible to override the default behavior (print error information and "crash") when an exception is raised, using `try`/`except` statements

```
try:
    strAge = input('Enter your age: ')
    intAge = int(strAge)
    print('You are {} years old.'.format(intAge))
except:
    print('Enter your age using digits 0-9!')
```

Custom behavior:

```
>>> ========== RESTART ==========
>>>
Enter your age: fifteen
Enter your age using digits 0-9!
>>>
```


Catching and handling exceptions

It is possible to override the default behavior (print error information and "crash") when an exception is raised, using `try`/`except` statements

If an exception is raised while executing the `try` block, then the block of the associated `except` statement is executed

```
try:
    strAge = input('Enter your age: ')
    intAge = int(strAge)
    print('You are {} years old.'.format(intAge))
except:
    print('Enter your age using digits 0-9!')
```

The `except` code block is the exception handler

Custom behavior:

```
>>> ========== RESTART ==========
>>>
Enter your age: fifteen
Enter your age using digits 0-9!
>>>
```

Format of a `try/except` statement pair

The format of a `try`/`except` pair of statements is:

```
try:
    <indented code block>
except:
    <exception handler block>
<non-indented statement>
```

The exception handler handles any exception raised in the `try` block

The `except` statement is said to catch the (raised) exception

It is possible to restrict the `except` statement to catch exceptions of a specific type only

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
<non-indented statement>
```


Format of a `try/except` statement pair

```
def readAge(filename):
    'converts first line of file filename to an integer and prints it'
    try:
        infile = open(filename)
        strAge = infile.readline()
        age = int(strAge)
        print('age is', age)
except ValueError:
        print('Value cannot be converted to integer.')
```

It is possible to restrict the `except` statement to catch exceptions of a specific type only

```
1 fifteen
```

age.txt

```
>>> readAge('age.txt')
Value cannot be converted to integer.
>>>
```

Format of a `try/except` statement pair

```
def readAge(filename):
    'converts first line of file filename to an integer and prints it'
    try:
        infile = open(filename)
        strAge = infile.readline()
        age = int(strAge)
        print('age is', age)
except ValueError:
        print('Value cannot be converted to integer.')
```

It is possible to restrict the `except` statement to catch exceptions of a specific type only

```
1 fifteen
```

age.txt

default exception handler prints this

```
>>> readAge('age.txt')
Value cannot be converted to integer.
>>> readAge('age.text')
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    readAge('age.text')
  File "/Users/me/ch7.py", line 12, in readAge
    infile = open(filename)
IOError: [Errno 2] No such file or directory: 'age.text'
>>>
```


Multiple exception handlers

```
def readAge(filename):
    'converts first line of file filename to an integer and prints it'
    try:
        infile = open(filename)
        strAge = infile.readline()
        age = int(strAge)
        print('age is',age)
    except IOError:
        # executed only if an IOError exception is raised
        print('Input/Output error.')
    except ValueError:
        # executed only if a ValueError exception is raised
        print('Value cannot be converted to integer.')
    except:
        # executed if an exception other than IOError or ValueError is raised
        print('Other error.')
```

Expensive Mistakes!

Maiden Flight of Ariane 5

On June 4, 1996, the Ariane 5 rocket developed over many years by the European Space Agency flew its first test flight. Seconds after the launch, the rocket exploded.

The crash happened when an overflow exception got raised during a conversion from floating point to integer. The cause of the crash was not the unsuccessful conversion (it turns out that it was of no consequence); the real cause was that the exception was not handled. Because of this, the rocket control software crashed and shut the rocket computer down. Without its navigation system, the rocket started turning uncontrollably, and the onboard monitors made the rocket self-destruct.

This was probably one of the most expensive computer bugs in history.

Controlling the exceptional control flow

```
>>> try:
    f(2)
except:
    print('!!')

Start f
Start g
Start h
!!
```

```
n = 2
print('Start f')
g(n-1)
...
print(n)
```
f(2)

```
n = 1
print('Start g')
h(n-1)
...
print(n)
```
g(1)

```
n = 0
print('Start h')
print(1/n)
print(n)
```
h(0)

```
1. def h(n):
2.     print('Start h')
3.     print(1/n)
4.     print(n)
5.
6. def g(n):
7.     print('Start g')
8.     h(n-1)
9.     print(n)
10.
11. def f(n):
12.     print('Start f')
13.     g(n-1)
14.     print(n)
```

Modules, revisited

A module is a file containing Python code.

When the module is executed (imported), then the module is (also) a namespace.

- This namespace has a name, typically the name of the module.
- In this namespace live the names that are defined in the global scope of the module: the names of functions, values, and classes defined in the module.

Built-in function `dir()` returns the names defined in a namespace

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__name__', '__package__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor',
'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan',
'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>>
```


Modules, revisited

A module is a file containing Python code.

When the module is executed (imported), then the module is (also) a namespace.

- This namespace has a name, typically the name of the module.
- In this namespace live the names that are defined in the global scope of the module: the names of functions, values, and classes defined in the module.
- These names are the module's attributes.

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__name__', '__package__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor',
'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan',
'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>>
```

Modules, revisited

A module is a file containing Python code.

When the module is executed (imported), then the module is (also) a namespace.

- This namespace has a name, typically the name of the module.
- In this namespace live the names that are defined in the global scope of the module: the names of functions, values, and classes defined in the module.
- These names are the module's attributes.

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__name__', '__package__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor',
'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan',
'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>> math.sqrt
<built-in function sqrt>
>>> math.pi
3.141592653589793
```

To access the imported module's attributes, the name of the namespace must be specified