Case Study: Python

Sequence Control

Dr. Nguyen Hua Phung

HCMC University of Technology, Viet Nam

08, 2020

Outline

Expressions

Precedence

Operator	Meaning	Arity	Assoc.
if – else	Conditional	Ternary	???
or	Boolean OR	Binary	Left
and	Boolean AND	Binary	Left
not x	Boolean NOT	Unary	Right
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparisons, including membership tests and identity tests	Binary	None
+, -	Addition and subtraction	Binary	Left
*, @, /, //, %	Multiplication, matrix multiplication, division, floor division, remainder	Binary	Left
+x, -x	Positive, negative	Unary	Right
**	Exponent	Binary	Right

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

Associativity

- **None-associativity**
  x op y

Associativity

- **None-associativity**
  x op y <= Ok

Associativity

- **None-associativity**
  x op y op z

Associativity

- **None-associativity**
  x op y op z <= wrong
  x < y

Associativity

- **None-associativity**
  x op y
  x < y <= Ok

Associativity

- **None-associativity**
  x op y
  x < y < z

Associativity

- **None-associativity**
  x op y
  x < y < z <= wrong

Associativity

- **None-associativity**
  x op y
  Python: x < y < z <= Ok

Associativity

- **None-associativity**
  x op y
  Python: x < y < z => x < y and y < z

Associativity

- **None-associativity**
  Python: x op y op z => x op y and y op z
  Python: x < y < z => x < y and y < z

Associativity

- **None-associativity**
- **if expression**

Associativity

- **None-associativity**
- **if expression**

<exp1> **if** <exp2> **else** <exp3>

Associativity

- **None-associativity**
- **if expression**

<exp1> **if** <exp2> **else** <exp3>

if <exp2>: **return** <exp1>
else: **return** <exp3>

Associativity

- **None-associativity**
- **if expression**

<exp1> **if** <exp2> **else** <exp3>

y = 3 **if** x > 1 **else** 4

Associativity

- **None-associativity**
- **if expression**

<exp1> **if** <exp2> **else** <exp3>

<exp1> **if** <exp2> **else** <exp3> **if** <exp5> **else** <exp6>

Associativity

- **None-associativity**
- **if expression**

<exp1> **if** <exp2> **else** <exp3>

[<exp1> **if** <exp2> **else** <exp3>] **if** <exp5> **else** <exp6>

Associativity

- **None-associativity**
- **if expression**

<exp1> **if** <exp2> **else** <exp3>

<exp1> **if** <exp2> **else** [<exp3> **if** <exp5> **else** <exp6>]

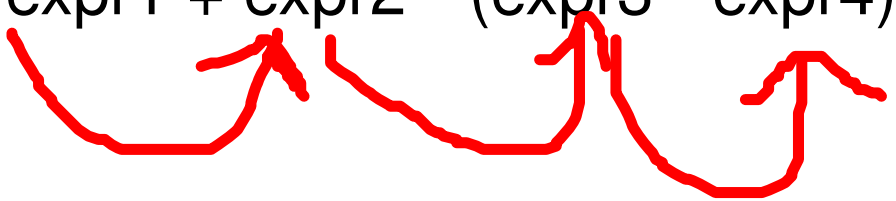
Evaluation order

- Evaluation order: from left to right

Evaluation order

- Evaluation order: from left to right

expr1 + expr2 * (expr3 - expr4)



Evaluation order

- Evaluation order: from left to right
- Short-circuit evaluation: and, or

Evaluation order

- Evaluation order: from left to right
- Short-circuit evaluation: and, or
  len(a) > 0 **and** a[0] != 1

Evaluation order

- Evaluation order: from left to right
- Short-circuit evaluation: and, or
- Method vs. Operator

Evaluation order

- Evaluation order: from left to right
- Short-circuit evaluation: and, or
- Method vs. Operator
  $obj_1.\_\_add\_\_(obj_2) \equiv obj_1 + obj_2$
  $obj_1.\_\_sub\_\_(obj_2) \equiv obj_1 - obj_2$

Statements

For statement

- **Iterable**: includes method **__iter__(self)** which returns **Iterator**

For statement

- **Iterable**: includes method **__iter__(self)** which returns **Iterator**
- **Iterator**: includes method **__next__(self)** which returns an item or raises StopIteration if no more item.

For statement

- **Iterable**: includes method **__iter__(self)** which returns **Iterator**
- **Iterator**: includes method **__next__(self)** which returns an item or raises StopIteration if no more item.
- meaning of **for** statement:

for <var> **in** <expression>:
  <stmt_list>

For statement

- **Iterable**: includes method **__iter__(self)** which returns **Iterator**
- **Iterator**: includes method **__next__(self)** which returns an item or raises StopIteration if no more item.
- meaning of **for** statement:

```
for <var> in <expression>:
    <stmt_list>
```

```
<tmp> = iter(<expression>)
while True:
    try:
        <var> = next(<tmp>)
        <stmt_list>
    except StopIteration:
        break
```

For statement

- **Iterable**: includes method **__iter__(self)** which returns **Iterator**
- **Iterator**: includes method **__next__(self)** which returns an item or raises StopIteration if no more item.
- meaning of **for** statement:

```
for <var> in <expression>:
    <stmt_list>
else:
    <stmt_list2>
```

```
<tmp> = iter(<expression>)
while True:
    try:
        <var> = next(<tmp>)
        <stmt_list>
    except StopIteration:
        break
```

Subprograms

Subprogram Mechanisms

- Simple Call Return
- Recursive Call
- Exception Processing Handler
- Coroutine
- Scheduled Subprograms
- Tasks

Recursive Call

Exception Handler

- Raise exceptions

  raise <Exception>
  assert <condition>(, <error message>)?

Exception Handler

- Raise exceptions

  raise <Exception> <= derived from *Exception*
  assert <condition>(, <error message>)?

Exception Handler

- Raise exceptions

  raise <Exception> <= derived from *Exception*
  assert <condition>(, <error message>)?

- Handling exceptions

  try:
  <stmt_list>
  (**except** (<Exception> (as ID)?)?:
  <stmt_list>)+
  (**else**:
  <stmt_list>)?
  (**finally**:
  <stmt_list>)?

Exception Handler

- Raise exceptions

  raise <Exception> <= derived from *Exception*
  assert <condition>(, <error message>)?

- Handling exceptions

  try:
  <stmt_list> <= might raise exceptions
  (**except** (<Exception> (as ID)?)?:
  <stmt_list>)+
  (**else**:
  <stmt_list>)?
  (**finally**:
  <stmt_list>)?

Exception Handler

- Raise exceptions

  raise <Exception> <= derived from *Exception*
  assert <condition>(, <error message>)?

- Handling exceptions

  try:
    <stmt_list> <= might raise exceptions
  (**except** (<Exception> (as ID)?)?:
    <stmt_list>)+ <= process if exception caught
  (**else**:
    <stmt_list>)?
  (**finally**:
    <stmt_list>)?

Exception Handler

- Raise exceptions

  raise <Exception> <= derived from *Exception*
  assert <condition>(, <error message>)?

- Handling exceptions

  try:
  <stmt_list> <= might raise exceptions
  (**except** (<Exception> (as ID)?)?:
  <stmt_list>)+ <= process if exception caught
  (**else**:
  <stmt_list>)? <= executed if no exception raised
  (**finally**:
  <stmt_list>)?

Exception Handler

- Raise exceptions

  raise <Exception> <= derived from *Exception*
  assert <condition>(, <error message>)?

- Handling exceptions

  try:
  <stmt_list> <= might raise exceptions
  (**except** (<Exception> (as ID)?)?:
  <stmt_list>)+ <= process if exception caught
  (**else**:
  <stmt_list>)? <= executed if no exception raised
  (**finally**:
  <stmt_list>)? <= always executed

Example

```
def linux_interaction():
    assert ('linux' in sys.platform), \
                "Function can only run on Linux systems."
    print('Doing something.')

try:
    linux_interaction()
except AssertionError as error:
    print(error)
else:
    try:
        with open('file.log') as file:
            read_data = file.read()
    except FileNotFoundError as fnf_error:
        print(fnf_error)
finally:
    print('Cleaning up, irrespective of any exceptions.')
```

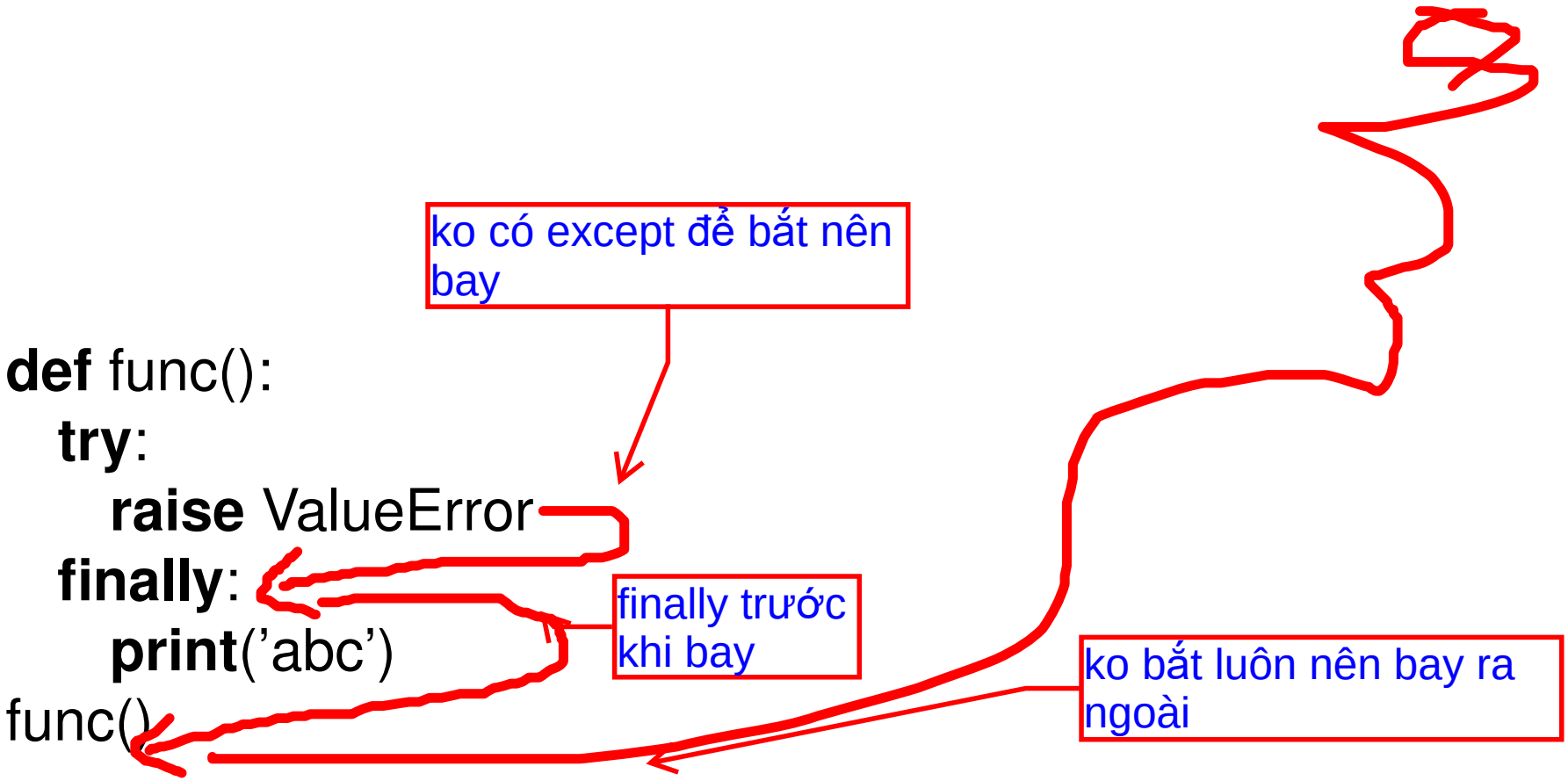
[https://realpython.com/python-exceptions/]

Finally Block

- If an exception is not handled by an **except** clause, the exception is re-raised **after** the **finally** clause has been executed.

Finally Block

- If an exception is not handled by an **except** clause, the exception is re-raised **after** the **finally** clause has been executed.



Finally Block

- If an exception is not handled by an **except** clause, the exception is re-raised **after** the **finally** clause has been executed.

```
def func():
    try:
        raise ValueError
    finally:
        print('abc')
```

func() => abc => ValueError

Finally Block

- If an exception is not handled by an **except** clause, the exception is re-raised **after** the **finally** clause has been executed.
- An exception raised in **except** or **else** clause is re-raised **after** the **finally** clause has been executed.

Finally Block

- If an exception is not handled by an **except** clause, the exception is re-raised **after** the **finally** clause has been executed.
- An exception raised in **except** or **else** clause is re-raised **after** the **finally** clause has been executed.
- The **finally** clause will execute **just prior** to the **break**, **continue** or **return** statement's execution in **try** clause

Finally Block

- If an exception is not handled by an **except** clause, the exception is re-raised **after** the **finally** clause has been executed.
- An exception raised in **except** or **else** clause is re-raised **after** the **finally** clause has been executed.
- The **finally** clause will execute **just prior** to the **break**, **continue** or **return** statement's execution in **try** clause
- A **return** in **finally** clause $>>$ **return** in **try** clause.

Finally Block

- If an exception is not handled by an **except** clause, the exception is re-raised **after** the **finally** clause has been executed.
- An exception raised in **except** or **else** clause is re-raised **after** the **finally** clause has been executed.
- The **finally** clause will execute **just prior** to the **break**, **continue** or **return** statement's execution in **try** clause
- A **return** in **finally** clause >> **return** in **try** clause.

```
def func():
    try:
        return True
    finally:
        return False
func()
```

Finally Block

- If an exception is not handled by an **except** clause, the exception is re-raised **after** the **finally** clause has been executed.
- An exception raised in **except** or **else** clause is re-raised **after** the **finally** clause has been executed.
- The **finally** clause will execute **just prior** to the **break**, **continue** or **return** statement's execution in **try** clause
- A **return** in **finally** clause >> **return** in **try** clause.

```
def func():
    try:
        return True
    finally:
        return False
func() => False
```

Coroutines

- Generators like functions but **yield** instead of **return**

```
def gen():
    yield 1
    yield 2
    yield 3
```

```
x = gen()
next(x)
next(x)
next(x)
```

Coroutines

- Generators like functions but **yield** instead of **return**

```
def gen():
    yield 1
    yield 2
    yield 3
```

```
x = gen()
next(x)
next(x)
next(x)
```

Coroutines

- Generators like functions but **yield** instead of **return**

```
def gen():
    yield 1
    yield 2
    yield 3
```

```
x = gen() => x->generator
next(x)
next(x)
next(x)
```

Coroutines

- Generators like functions but **yield** instead of **return**

```
def gen():
    yield 1
    yield 2
    yield 3
```

```
x = gen()
next(x)
next(x)
next(x)
```

Coroutines

- Generators like functions but **yield** instead of **return**

```
def gen():
    yield 1
    yield 2
    yield 3
```

```
x = gen()
next(x)
next(x)
next(x)
```

Coroutines

- Generators like functions but **yield** instead of **return**

```
def gen():
    yield 1
    yield 2
    yield 3
```

```
x = gen()
next(x) => 1
next(x)
next(x)
```

Coroutines

- Generators like functions but **yield** instead of **return**

```
def gen():
    yield 1
    yield 2
    yield 3
```

```
x = gen()
next(x)
next(x)
next(x)
```

Coroutines

- Generators like functions but **yield** instead of **return**

```
def gen():
    yield 1
    yield 2
    yield 3
```

```
x = gen()
next(x)
next(x)
next(x)
```

Coroutines

- Generators like functions but **yield** instead of **return**

```
def gen():
    yield 1
    yield 2
    yield 3
```

```
x = gen()
next(x)
next(x) => 2
next(x)
```

Coroutines

- Generators like functions but **yield** instead of **return**

```
def gen():
    yield 1
    yield 2
    yield 3
```

```
x = gen()
next(x)
next(x)
next(x)
```

Coroutines

- Generators like functions but **yield** instead of **return**

```
def gen():
    yield 1
    yield 2
    yield 3
```

```
x = gen()
next(x)
next(x)
next(x)
```

Coroutines

- Generators like functions but **yield** instead of **return**

```
def gen():
    yield 1
    yield 2
    yield 3
```

```
x = gen()
next(x)
next(x)
next(x) => 3
```

Coroutines

- Generators like functions but **yield** instead of **return**

```
def gen():
    yield 1
    yield 2
    yield 3
```

```
x = gen()
next(x)
next(x)
next(x)
```

- Generators are also created by comprehension

```
gen = (i for i in range(1,4))
```

Coroutines

- Generators like functions but **yield** instead of **return**

```
def gen():
    yield 1
    yield 2
    yield 3
```

```
x = gen()
next(x)
next(x)
next(x)
```

- Generators are also created by comprehension

  gen = (i **for** i **in range**(1,4)) => gen->generator

Coroutines

- Generators like functions but **yield** instead of **return**

```
def gen():
    yield 1
    yield 2
    yield 3
```

```
x = gen()
next(x)
next(x)
next(x)
```

- Generators are also created by comprehension

  gen = (i **for** i **in range**(1,4))

- used for large data. E.g., read the content of a file.

Coroutines

- Generators like functions but **yield** instead of **return**

```
def gen():
   yield 1
   yield 2
   yield 3
```

```
x = gen()
next(x)
next(x)
next(x)
```

- Generators are also created by comprehension

  gen = (i **for** i **in range**(1,4))

- used for large data. E.g., read the content of a file.

```
def csv_reader(fname):
    file = open(fname)
    result = file.read().split("\n")
    return result
len(csv_reader('abc.csv'))
```

Coroutines

- Generators like functions but **yield** instead of **return**

```
def gen():
    yield 1
    yield 2
    yield 3
```

```
x = gen()
next(x)
next(x)
next(x)
```

- Generators are also created by comprehension

```
gen = (i for i in range(1,4))
```

- used for large data. E.g., read the content of a file.

```
def csv_reader(fname):
    file = open(fname)
    result = file.read().split("\n")
    return result
len(csv_reader('abc.csv'))
```

```
def csv_reader(fname):
    for row in open(fname, "r"):
        yield row
sum(1 for _ in
        csv_reader('abc.csv'))
```

Scheduled Subprograms

- module **schedule**

```
import schedule
import time

def job():
    print("I'm working...")

schedule.every(10).minutes.do(job)
schedule.every().hour.do(job)
schedule.every().day.at("10:30").do(job)

while True:
    schedule.run_pending()
    time.sleep(1)
```

Scheduled Subprograms

- module **schedule**

```
import schedule
import time

def job():
    print("I'm working...")

schedule.every(10).minutes.do(job) => register scheduled job
schedule.every().hour.do(job)
schedule.every().day.at("10:30").do(job)

while True:
    schedule.run_pending()
    time.sleep(1)
```

Scheduled Subprograms

- module **schedule**

```
import schedule
import time

def job():
    print("I'm working...")

schedule.every(10).minutes.do(job) => register scheduled job
schedule.every().hour.do(job)
schedule.every().day.at("10:30").do(job)

while True:
    schedule.run_pending() => call scheduled job
    time.sleep(1)
```

- module **threading** and module **multiprocessing** and some other modules

Tasks

- module **threading** and module **multiprocessing** and some other modules
- threading.Thread(target=<function>) or multiprocessing.Process(target=<function>)

Tasks

- module **threading** and module **multiprocessing** and some other modules
- threading.Thread(target=<function>) or multiprocessing.Process(target=<function>)
- start() => call <function> run on another thread or another process

Tasks

- module **threading** and module **multiprocessing** and some other modules
- threading.Thread(target=<function>) or multiprocessing.Process(target=<function>)
- start() => call <function> run on another thread or another process
- join() => wait for <function> stop.

Example

```
import os
import time
import threading
import multiprocessing

NUM_WORKERS = 4

def only_sleep():
    """ Do nothing, wait for a timer to expire """
    print("PID: %s, Process Name: %s, Thread Name: %s" % (
        os.getpid(),
        multiprocessing.current_process().name,
        threading.current_thread().name)
    )
    time.sleep(1)
```

[https://code.tutsplus.com/articles/introduction-to-parallel-and-concurrent-programming-in-python--cms-28612]

Example

```
## Run tasks serially
start_time = time.time()
for _ in range(NUM_WORKERS):
      only_sleep()
end_time = time.time()
print("Serial time=", end_time - start_time)

# Run tasks using threads
start_time = time.time()
threads = [threading.Thread(target=only_sleep)
                                        for _ in range(NUM_WORKERS)]
[thread.start() for thread in threads]
[thread.join() for thread in threads]
end_time = time.time()
print("Threads time=", end_time - start_time)
```

References I

[1] Python Tutorial, http:w3schools.com/python, 10 08 2020.

[2] Python Programming Language, https://www.geeksforgeeks.org/python-programming-language/, 10 08 2020.

[3] Python Tutorial, https://www.tutorialspoint.com/python, 10 08 2020.

[4] Introduction to Python 3, https://realpython.com/python-introduction/, 10 08 2020.