

Case Study: Python

Sequence Control

Dr. Nguyen Hua Phung

HCMC University of Technology, Viet Nam

08, 2020

1 Data Types

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 mul- tiplication, division, floor division, remainder	Binary	Left
+X, -X	Positive, negative	Unary	Right
**	Exponent	Binary	Right

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

- **None-associativity**

$x \text{ op } y$

- **None-associativity**

$x \text{ op } y \leq \text{Ok}$

- **None-associativity**

$x \text{ op } y \text{ op } z$

- **None-associativity**

$x \text{ op } y \text{ op } z \leq \text{wrong}$

$x < y$

- **None-associativity**

$x \text{ op } y$

$x < y \leq \text{Ok}$

- **None-associativity**

$x \text{ op } y$

$x < y < z$

- **None-associativity**

$x \text{ op } y$

$x < y < z$ **<= wrong**

- **None-associativity**

$x \text{ op } y$

Python: $x < y < z$ **<= Ok**

- **None-associativity**

$x \text{ op } y$

Python: $x < y < z \Rightarrow x < y \text{ and } y < z$

- **None-associativity**

Python: $x \text{ op } y \text{ op } z \Rightarrow x \text{ op } y \text{ and } y \text{ op } z$

Python: $x < y < z \Rightarrow x < y \text{ and } y < z$

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

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

`<exp1> if <exp2> else <exp3>`

- None-associativity
- if expression

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

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

<exp1> if <exp2> else <exp3>

y = 3 if x > 1 else 4

- None-associativity
- if expression

`<exp1> if <exp2> else <exp3>`

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

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

`<exp1> if <exp2> else <exp3>`

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

- None-associativity
- if expression

`<exp1> if <exp2> else <exp3>`

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

- Evaluation order: from left to right

- Evaluation order: from left to right
 $\text{expr1} + \text{expr2} * (\text{expr3} - \text{expr4})$

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

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

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

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

$\text{obj}_1._\text{add}_\text{(obj}_2) \equiv \text{obj}_1 + \text{obj}_2$

$\text{obj}_1._\text{sub}_\text{(obj}_2) \equiv \text{obj}_1 - \text{obj}_2$

Statements

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

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

- **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>
```

- **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
```


- **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

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

- Raise exceptions

raise <Exception>

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

- Raise exceptions

raise <Exception> *<= derived from Exception*

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

- 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>)?

- 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>)?

- 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>)?

- 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>)?

- 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*

```
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/>]

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

- 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()
```

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
```

- 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.

- 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

- 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.

- 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()
```

- 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
```

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

def gen():	x = gen()
yield 1	next(x)
yield 2	next(x)
yield 3	next(x)

- 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 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)
```

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

```
def gen():
```

```
    yield 1
```

```
    yield 2
```

```
    yield 3
```

```
x = gen()
```

```
next(x)
```

```
next(x)
```

```
next(x)
```

- 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)
```

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

def gen():	x = gen()
yield 1	next(x)
yield 2	next(x)
yield 3	next(x)

- 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)
```

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

def gen():	x = gen()
yield 1	next(x)
yield 2	next(x)
yield 3	next(x)

- 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
```


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

def gen():	x = gen()
yield 1	next(x)
yield 2	next(x)
yield 3	next(x)

- Generators are also created by comprehension

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

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

def gen():	x = gen()
yield 1	next(x)
yield 2	next(x)
yield 3	next(x)

- Generators are also created by comprehension

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

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

def gen():	x = gen()
yield 1	next(x)
yield 2	next(x)
yield 3	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.

- 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'))
```

- 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
```

```
def csv_reader(fname):
    for row in open(fname, "r"):
        yield row
```

```
len(csv_reader('abc.csv'))
```

```
sum(1 for _ in
    csv_reader('abc.csv'))
```

- 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)
```

- 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)
```

- 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

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

- 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

- 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]
```

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)
```

- [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.