Learning Python
(Advanced Topics)

DATAVORA

Hamdi NEHDI

Python: Advanced Topics

- System focused programming.
- Network programming.
- Python design patterns.
- Computer Science and Computer Linguistics.

Python: Advanced Topics

1. Serialization:

   - ✓ Definition: Arrange (something) in a series.
   - ✓ The process of translating data structures into a format that can be stored(for example a file) or transmitted (for example across network) and reconstructed later.
   - ✓ Example: Fetching JSON data from a RESTFUL API.

Python: Advanced Topics

1. Serialization:

   - JSON: (Javascript Object Notation)

     - ✓ A textual-based format (a lightweight data-interchange format), widely used as object serialization over the web.
     - ✓ Easy for machines to parse and generate (easy for humans to read and write as well).
     - ✓ A format that is completely language independent known by many PL such as: Python, C/C++, Java …

Python: Advanced Topics

1. Serialization:

   - JSON: (Javascript Object Notation)

     - ✓ JSON is built on two universal data structures:
       1. A collection of name/value pairs. This is realized as an object, record, dictionary or hash table.
       2. An ordered list of values. This is realized as an array, vector, list or sequence.

     - ✓ Python provides built-in JSON librairies to encode and decode JSON.

     - ✓ The process of encoding JSON is usually called serialization.

Python: Advanced Topics

1. Serialization:

   - JSON: (Javascript Object Notation)

```
simple = dict(int_list=[1, 2, 3],
              text='Level 1',
              number=3.44,
              boolean=True,
              none=None)
```

Python: Advanced Topics

1. Serialization:

   - JSON: (Javascript Object Notation)

```
import json

print json.dumps(simple)

{"text": “Level 1", "none": null, "boolean": true, "number": 3.44, "int_list": [1, 2, 3]}
```

Python: Advanced Topics

1. Serialization:
   - JSON: (Javascript Object Notation)

```
import json

print json.dumps(simple, indent=4)

{
    "text": “Level 1",
    "none": null,
    "boolean": true,
    "number": 3.44,
    "int_list": [
        1,
        2,
        3
    ]
}
```

```
from pprint import pprint
```

Python: Advanced Topics

1. Serialization:
   - JSON: (Javascript Object Notation)

```
from datetime import datetime

simple = dict(int_list=[1, 2, 3],
              text='Level 1',
              number=3.44,
              boolean=True,
              none=None,
              date=datetime(2018, 5, 25)
)
```

Python: Advanced Topics

1. Serialization:

   - JSON: (Javascript Object Notation)

```
import json

print json.dumps(simple, indent=4)

TypeError: Object of type 'datetime' is not JSON serializable
```

Python: Advanced Topics

1. Serialization:
   - JSON: (Javascript Object Notation)

```
from datetime import datetime

simple = dict(int_list=[1, 2, 3],
              text='Level 1',
              number=3.44,
              boolean=True,
              none=None,
              date=datetime(2018, 5, 25).isoformat()
)
```

Python: Advanced Topics

1. Serialization:
   - JSON: (Javascript Object Notation)

```
import json

print json.dumps(simple, indent=4)

{
    "text": “Level 1",
    "none": null,
    "boolean": true,
    "number": 3.44,
    "int_list": [
        1,
        2,
        3
    ]
    “date": “2018-05-25T00:00:00"
}
```

Python: Advanced Topics

1. Serialization:

   - How it works:

     - ✓ The json module offers two ways to serialize objects:

       1. Implement a default function that takes an object and returns something that JSONEncoder understands.
       2. Implement or subclass a JSONEncoder yourself and pass it as “cls” to the dump method.

Python: Advanced Topics

1. Serialization:
   - How it works:

```
from datetime import datetime
import json

class CustomEncoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, datetime):
            return {'__datetime__': o.replace(microsecond=0).isoformat()}
        return {'__{}__'.format(o.__class__.__name__): o.__dict__}
```

Python: Advanced Topics

1. Serialization:
   - How it works:

```
import json

print json.dumps(simple, indent=4, cls=CustomEncoder)

{
    "text": “Level 1",
    "none": null,
    "boolean": true,
    "number": 3.44,
    "int_list": [
        1,
        2,
        3
    ]
    “date": {
        “__datetime__": “2018-05-25T00:00:00“
    }
}
```

Python: Advanced Topics

1. Serialization:
   - How it works:

```
def serialize_my_object(obj):
    if isinstance(obj, datetime):
        return obj.isoformat() + “Z“
    elif hasattr(obj, “__class__”):
        return obj.__dict__
    elif isinstance(obj, Exception):
        return {
            "error": obj.__class__.__name__,
            "args": obj.args,
        }
    return str(obj)
```

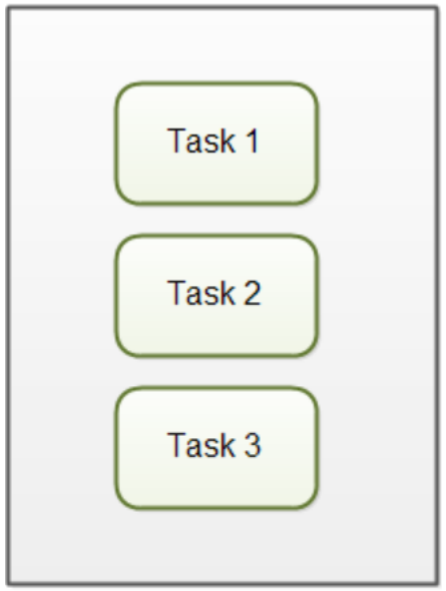
Python: Advanced Topics

2. Concurrency and Parallelism:

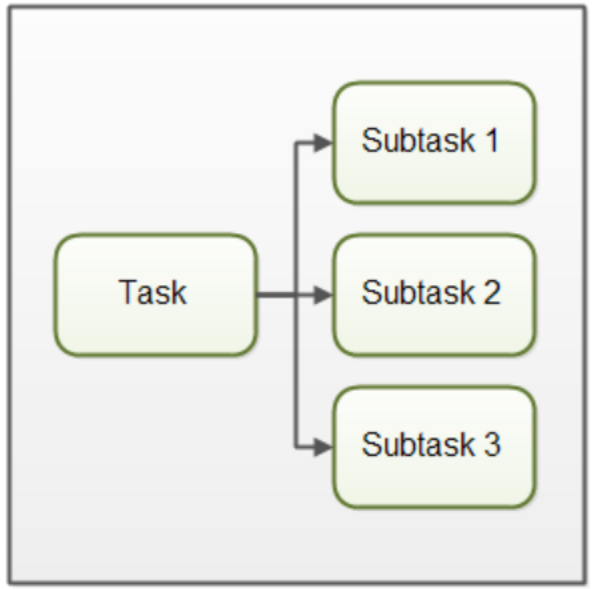
- ✓ Concurrency is when two or more tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant. For example, multitasking on a single-core machine.
- ✓ Parallelism is when tasks literally run at the same time, for example, on a multicore processor.
- ✓ An application can be concurrent, but not parallel. This means that it processes more than one task at the same time, but the tasks are not broken down into subtasks.
- ✓ An application can also be parallel but not concurrent. This means that the application only works on one task at a time, and this task is broken down into subtasks which can be processed in parallel.

Python: Advanced Topics

2. Concurrency and Parallelism:



Concurrency:
Multiple tasks makes progress at the same time.



Parallelism:
Each task is broken into subtasks which can be processed in parallel.

Python: Advanced Topics

2. Concurrency and Parallelism:

❖ Processes and Threads:

1) Process:

- ✓ A process is an instance of a program that is being executed. It contains the program code and its current activity.
- ✓ A process has a self-contained execution environment. It has a complete set of private basic run-time resources; in particular, each process has its own memory space.

Python: Advanced Topics

2. Concurrency and Parallelism:

❖ Processes and Threads:

1) Thread:

- ✓ A thread is like a mini-process (also called light-weight process) that lives inside a process.
- ✓ A thread shares memory space and efficiently reads and writes to the same variables.
- ✓ A thread does not require much memory overhead; it is cheaper than a process.
- ✓ Every process has at least one thread (usually many threads).

Python: Advanced Topics

2. Concurrency and Parallelism:

- Processes and Threads:

PROCESSES	THREADS
Processes don't share memory	Threads share memory
Spawning/switching processes is expensive	Spawning/switching threads is less expensive
Processes require more resources	Threads require fewer resources (are sometimes called lightweight processes)
No memory synchronisation needed	You need to use synchronisation mechanisms to be sure you're correctly handling the data

Python: Advanced Topics

2. Concurrency and Parallelism:

- ✓ Python is one of the most popular languages for data processing and data science in general. The ecosystem provides a lot of libraries and frameworks that facilitate high-performance computing. Doing parallel programming in Python can prove quite tricky, though.

- ✓ The Global Interpreter Lock (GIL) is a mutex that makes things thread-safe.

- ✓ Due to the GIL, we can't achieve true parallelism via multithreading. Basically, two different native threads of the same process can't run Python code at once.

```
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)


def crunch_numbers():
    """ Do some computations """
    print("PID: %s, Process Name: %s, Thread Name: %s" % (
        os.getpid(),
        multiprocessing.current_process().name,
        threading.current_thread().name)
    )
    x = 0
    while x < 10000000:
        x += 1
```

```
## 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)

# Run tasks using processes
start_time = time.time()
processes = [multiprocessing.Process(target=only_sleep()) for _ in range(NUM_WORKERS)]
[process.start() for process in processes]
[process.join() for process in processes]
end_time = time.time()

print("Parallel time=", end_time - start_time)
```

PID: 95726, Process Name: MainProcess, Thread Name: MainThread
PID: 95726, Process Name: MainProcess, Thread Name: MainThread
PID: 95726, Process Name: MainProcess, Thread Name: MainThread
PID: 95726, Process Name: MainProcess, Thread Name: MainThread
Serial time= 4.018089056015015

PID: 95726, Process Name: MainProcess, Thread Name: Thread-1
PID: 95726, Process Name: MainProcess, Thread Name: Thread-2
PID: 95726, Process Name: MainProcess, Thread Name: Thread-3
PID: 95726, Process Name: MainProcess, Thread Name: Thread-4
Threads time= 1.0047411918640137

PID: 95728, Process Name: Process-1, Thread Name: MainThread
PID: 95729, Process Name: Process-2, Thread Name: MainThread
PID: 95730, Process Name: Process-3, Thread Name: MainThread
PID: 95731, Process Name: Process-4, Thread Name: MainThread
Parallel time= 1.014023780822754

```
start_time = time.time()
for _ in range(NUM_WORKERS):
    crunch_numbers()
end_time = time.time()

print("Serial time=", end_time - start_time)

start_time = time.time()
threads = [threading.Thread(target=crunch_numbers) 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)


start_time = time.time()
processes = [multiprocessing.Process(target=crunch_numbers) for _ in range(NUM_WORKERS)]
[process.start() for process in processes]
[process.join() for process in processes]
end_time = time.time()

print("Parallel time=", end_time - start_time)
```

PID: 96285, Process Name: MainProcess, Thread Name: MainThread
PID: 96285, Process Name: MainProcess, Thread Name: MainThread
PID: 96285, Process Name: MainProcess, Thread Name: MainThread
PID: 96285, Process Name: MainProcess, Thread Name: MainThread
Serial time= 2.705625057220459
PID: 96285, Process Name: MainProcess, Thread Name: Thread-1
PID: 96285, Process Name: MainProcess, Thread Name: Thread-2
PID: 96285, Process Name: MainProcess, Thread Name: Thread-3
PID: 96285, Process Name: MainProcess, Thread Name: Thread-4
Threads time= 2.6961309909820557
PID: 96289, Process Name: Process-1, Thread Name: MainThread
PID: 96290, Process Name: Process-2, Thread Name: MainThread
PID: 96291, Process Name: Process-3, Thread Name: MainThread
PID: 96292, Process Name: Process-4, Thread Name: MainThread
Parallel time= 0.8014059066772461

Python: Advanced Topics

2. Concurrency and Parallelism:

- Conclusion:

  - ✓ Multiprocessing provides a very similar interface to threading but for processes rather than threads.
  - ✓ Remember that processes achieve true parallelism, but they are more expensive to create.
  - ✓ Remember that a process can have more threads running inside it.
  - ✓ Do not mistake parallel for concurrent. Remember that only the parallel approach takes advantage of multi-core processors, whereas concurrent programming intelligently schedules tasks so that waiting on long-running operations is done while in parallel doing actual computation.

Python: Advanced Topics

2. UnitTest:
   - Some general rules of testing:
     - A testing unit should focus on one tiny bit of functionality and prove it correct.
     - Each test unit must be fully independent. Each test must be able to run alone, and also within the test suite, regardless of the order that they are called.
     - Try hard to make tests that run fast. If one single test needs more than a few milliseconds to run, development will be slowed down or the tests will not be run as often as is desirable.
     - Always run the full test suite before a coding session, and run it again after. This will give you more confidence that you did not break anything in the rest of the code.
     - It is a good idea to implement a hook that runs all tests before pushing code to a shared repository.
     - Use long and descriptive names for testing functions.

Python: Advanced Topics

2. UnitTest:
   - ✓ unittest is the batteries-included test module in the Python standard library.
   - ✓ Creating test cases is accomplished by subclassing unittest.TestCase.

```
import unittest

def fun(x):
    return x + 1

class MyTest(unittest.TestCase):
    def test_fun_3_4(self):
        self.assertEqual(fun(3), 4)

if __name__ == '__main__':
    unittest.main()
```

Python: Advanced Topics

2. UnitTest:

```
python simple_test.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Python: Advanced Topics

2. UnitTest:

```
python simple_test.py –v

test_fun_3_4 (__main__.MyTest) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

Python: Advanced Topics

2. UnitTest:

- Tests have 3 possible outcomes:
  - ✓ Ok ----→ The test passes.
  - ✓ FAIL ----→ The test does not pass, and raises an AssertionError exception.
  - ✓ ERROR ----→ The test raises an exception other than AssertionError.

Python: Advanced Topics

2. UnitTest:

```
import unittest

class OutcomesTest(unittest.TestCase):
    def test_pass(self):
        self.assertTrue(True)
    def test_fail(self):
        self.assertTrue(False)
    def test_error(self):
        raise RuntimeError('Test error!')

if __name__ == '__main__':
    unittest.main()
```

```
python test_truth.py -v

test_error (__main__.OutcomesTest) ... ERROR
test_fail (__main__.OutcomesTest) ... FAIL
test_pass (__main__.OutcomesTest) ... ok

======================================================================
ERROR: test_error (__main__.OutcomesTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_truth.py", line 12, in test_error
    raise RuntimeError('Test error!')
RuntimeError: Test error!

======================================================================
FAIL: test_fail (__main__.OutcomesTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_truth.py", line 9, in test_fail
    self.assertTrue(False)
AssertionError: False is not true

----------------------------------------------------------------------
Ran 3 tests in 0.003s

FAILED (failures=1, errors=1)
```

Python: Advanced Topics

2. UnitTest:
   - Common Assertions:

```
assertTrue(x, msg=None)
assertFalse(x, msg=None)
assertIsNone(x, msg=None)
assertIsNotNone(x, msg=None)
assertEqual(a, b, msg=None)
assertNotEqual(a, b, msg=None)
assertIs(a, b, msg=None)
assertIsNot(a, b, msg=None)
assertIn(a, b, msg=None)
assertNotIn(a, b, msg=None)
assertIsInstance(a, b, msg=None)
assertNotIsInstance(a, b, msg=None)
```

Python: Advanced Topics

2. UnitTest:
   - Other Assertions:

```
assertGreater(a, b, msg=None)
assertGreaterEqual(a, b, msg=None)
assertLess(a, b, msg=None)
assertLessEqual(a, b, msg=None)
assertRegex(text, regexp, msg=None)
assertSequenceEqual(a, b, msg=None)
```

```
assertListEqual(a, b, msg=None)
assertTupleEqual(a, b, msg=None)
assertSetEqual(a, b, msg=None)
assertDictEqual(a, b, msg=None)
```

Python: Advanced Topics

2. UnitTest:

- Testing for exceptions:

```
import unittest

def raises_error(*args, **kwds):
    raise ValueError('Invalid value: %s%s' % (args, kwds))

class ExceptionTest(unittest.TestCase):
    def test_trap_locally(self):
        try:
            raises_error('a', b='c')
        except ValueError:
            pass
        else:
            self.fail('Did not see ValueError')
    def test_assert_raises(self):
        self.assertRaises(ValueError, raises_error, 'a', b='c')

if __name__ == '__main__':
    unittest.main()
```

Python: Advanced Topics

2. UnitTest:
   - Testing for exceptions:

```
python test_exception.py –v

test_assert_raises (__main__.ExceptionTest) ... ok
test_trap_locally (__main__.ExceptionTest) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

Thank you for your attention