Moving to Python 3

A.B.C

2.7.15

3.6.5

Python 2 Current Situation

- Python 2 maintenance is going to be stopped in 2020(PEP 373).
  - The latest Python 2 is **Python 2.7**
  - Python 2.7 latest version: **2.7.15 2018-05-01**
  - Planned future release dates: **2.7.16 late 2018 - early 2019**
- There will be no Python 2.8 (see PEP 404).
  - Rule number six: there is ***no*** official Python 2.8 release. There **never** will be an official Python 2.8 release. It is an ex-release. Python 2.7 is the end of the Python 2 line of development.

Python 3 Current Situation (2018-7)

- Python 3.0 was released 2008
- 3.0.x, 3.1.x,3.2.x, 3.3.x **end-of-life**
- Latest: 3.7.0, 3.6.6, 3.5.5, 3.4.8
- Plan: 3.8.0 2019-10-20 (PEP 569)

Backwards-incompatible

- Python 2 -> Python 3
- Python 3 -> Python 4?

Python 3 Adoption

- https://www.jetbrains.com/research/python-developers-survey-2017/#python-3-adoption
- More and more project will only support python 3 with new version(https://python3statement.org/)
- Python 3 Readiness http://py3readiness.org/

Moving to Python 3

- Writing Python 2 and Python 3 compatible code (保守)
- Porting Python 2 code to Python 3 and drop python 2

Best Practise

- Only worry about supporting Python 2.7
- Make sure you have good test coverage (coverage.py can help; pip install coverage)
- Learn the differences between Python 2 & 3
- Use Futurize (or Modernize) to update your code (e.g. pip install future)
- Use Pylint to help make sure you don’t regress on your Python 3 support (pip install pylint)
- Use caniusepython3 to find out which of your dependencies are blocking your use of Python 3 (pip install caniusepython3)
- Once your dependencies are no longer blocking you, use continuous integration to make sure you stay compatible with Python 2 & 3 (tox can help test against multiple versions of Python; pipinstall tox)
- Consider using optional static type checking to make sure your type usage works in both Python 2 & 3 (e.g. use mypy to check your typing under both Python 2 & Python 3).

NEW FEATURE

PEP 238 -- Changing the Division Operator

https://www.python.org/dev/peps/pep-0238/

True Division

- **x/y** to return a reasonable approximation of the mathematical result of the division

```
>>> 3 / 4  # python 3
0.75
```

```
>>> 3 / 4  # python 2
0
```

```
>>> 3 / 4.0    # in python 2, we must use float type
0.75
```

Classic Division

- **x//y** to return the floor ("floor division"). We call the current, mixed meaning of x/y "classic division". If both x and y are integer, the result will be an integer, if at least one of them is float, the result will be a float.

```
>>> 3 // 4
0
>>> 4 // 3
1
>>> 3 // 4.0
0.0
>>> 4 // 3.0
1.0
```

Same results under Python 2 and Python 3

Same operator with different results

- **Classic division** will remain the default in the Python 2.x series; **true division** will be standard in Python 3.0.
- The // operator will be available to request floor division unambiguously.
- The future division statement, spelled **from __future__ import division**, will change the / operator to mean true division throughout the module.
- A command line option will enable run-time warnings for classic division applied to int or long arguments; another command line option will make true division the default.
- The standard library will use the future division statement and the // operator when appropriate, so as to completely avoid classic division.

PEP 237 -- Unifying Long Integers and Integers

https://www.python.org/dev/peps/pep-0237/

Long integer and integer in python 2

- Integer
  - There is a maximun integer **sys.maxint** (depends on your system, if our system is 64bit, the sys.maxint is 2^63 – 1, if our system is 32bit, the sys.maxint is 2^31- 1)
  - The size in memory is **fixed** (for example 24 bytes)
- Long integer
  - The size in memory is **dynamic** (from 28 bytes to the limit of system memory)

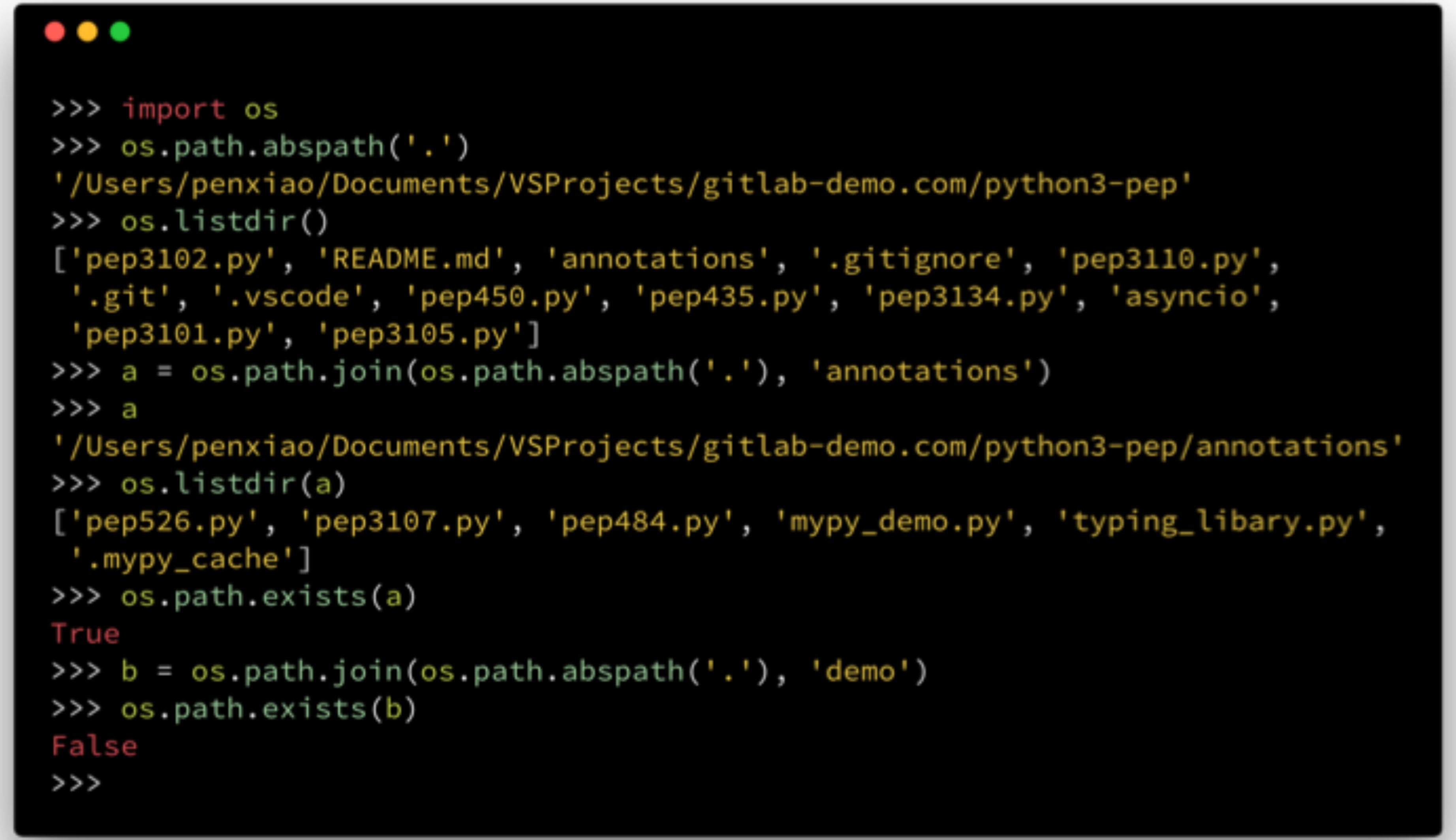
Python 3: only have integer

- Python 3 doesn’t have long integer, only have integer
- The python 3 integer behavour like python 2 long integer, both have a dynamic length.

PEP 428 -- The pathlib module -- object-oriented filesystem paths (Python 3.4)

https://www.python.org/dev/peps/pep-0428/

os.path



```
>>> import os
>>> os.path.abspath('.')
'/Users/penxiao/Documents/VSProjects/gitlab-demo.com/python3-pep'
>>> os.listdir()
['pep3102.py', 'README.md', 'annotations', '.gitignore', 'pep3110.py',
 '.git', '.vscode', 'pep450.py', 'pep435.py', 'pep3134.py', 'asyncio',
 'pep3101.py', 'pep3105.py']
>>> a = os.path.join(os.path.abspath('.'), 'annotations')
>>> a
'/Users/penxiao/Documents/VSProjects/gitlab-demo.com/python3-pep/annotations'
>>> os.listdir(a)
['pep526.py', 'pep3107.py', 'pep484.py', 'mypy_demo.py', 'typing_libary.py',
 '.mypy_cache']
>>> os.path.exists(a)
True
>>> b = os.path.join(os.path.abspath('.'), 'demo')
>>> os.path.exists(b)
False
>>>
```

PEP 435 -- Adding an Enum type to the Python standard library (Python 3.4)

https://www.python.org/dev/peps/pep-0435/

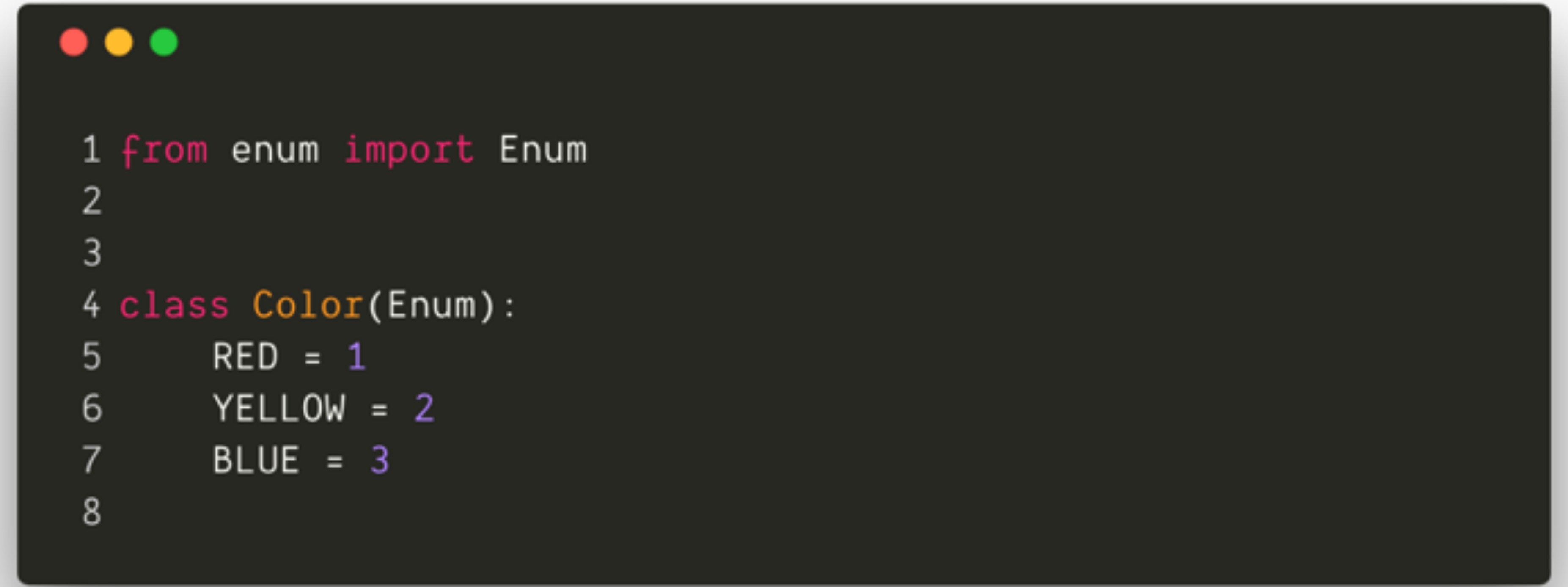
An enumeration is a set of symbolic names bound to unique, constant values. Within an enumeration, the values can be **compared** by identity, and the enumeration itself can be **iterated** over.

-- PEP 435

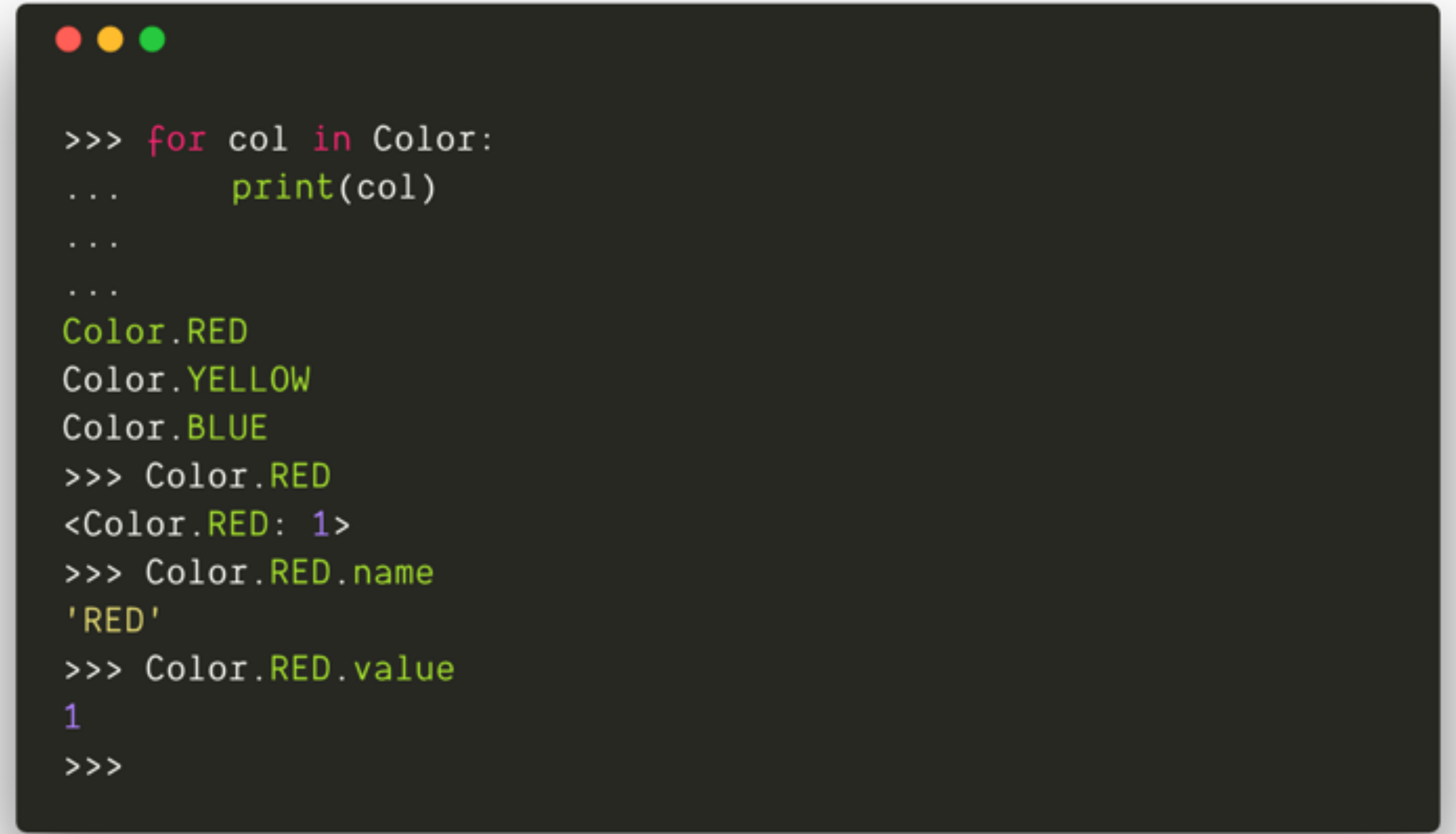
Before Enumeration

```
$ more contans.py

# define some contants
RED = 1
YELLOW = 2
BLUE = 3
```

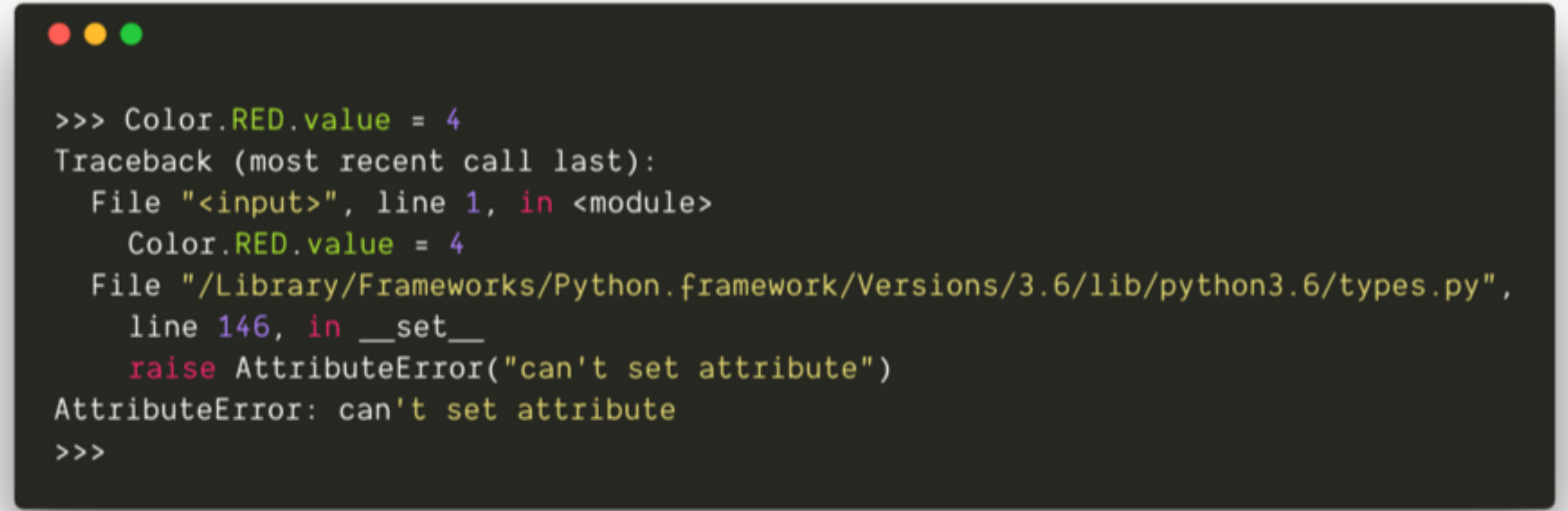
After Enumeration



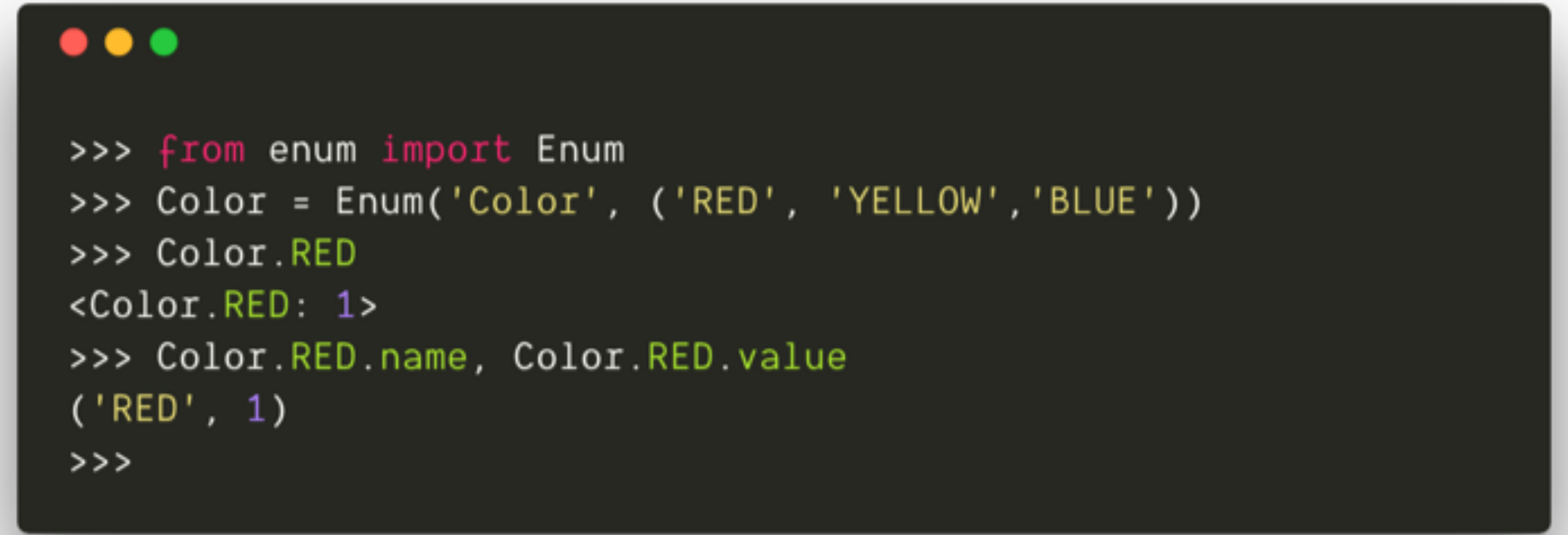
```python
from enum import Enum


class Color(Enum):
    RED = 1
    YELLOW = 2
    BLUE = 3

```



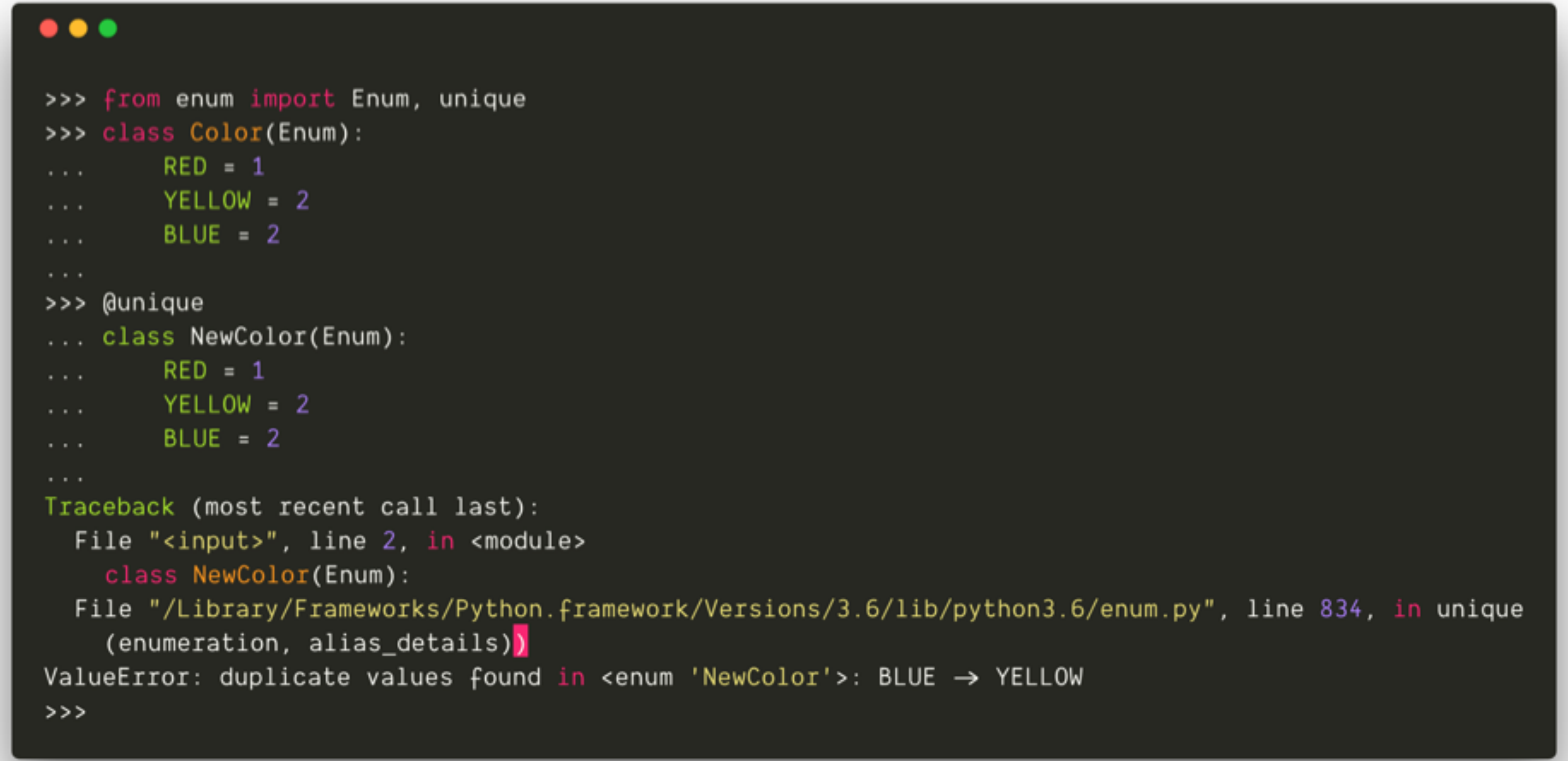
```
>>> for col in Color:
...     print(col)
...
...
Color.RED
Color.YELLOW
Color.BLUE
>>> Color.RED
<Color.RED: 1>
>>> Color.RED.name
'RED'
>>> Color.RED.value
1
>>>
```



```
>>> Color.RED.value = 4
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    Color.RED.value = 4
  File "/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/types.py",
    line 146, in __set__
    raise AttributeError("can't set attribute")
AttributeError: can't set attribute
>>>
```



```
>>> from enum import Enum
>>> Color = Enum('Color', ('RED', 'YELLOW','BLUE'))
>>> Color.RED
<Color.RED: 1>
>>> Color.RED.name, Color.RED.value
('RED', 1)
>>>
```



```
>>> from enum import Enum, unique
>>> class Color(Enum):
...     RED = 1
...     YELLOW = 2
...     BLUE = 2
...
>>> @unique
... class NewColor(Enum):
...     RED = 1
...     YELLOW = 2
...     BLUE = 2
...
Traceback (most recent call last):
  File "<input>", line 2, in <module>
    class NewColor(Enum):
  File "/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/enum.py", line 834, in unique
    (enumeration, alias_details))
ValueError: duplicate values found in <enum 'NewColor'>: BLUE -> YELLOW
>>>
```

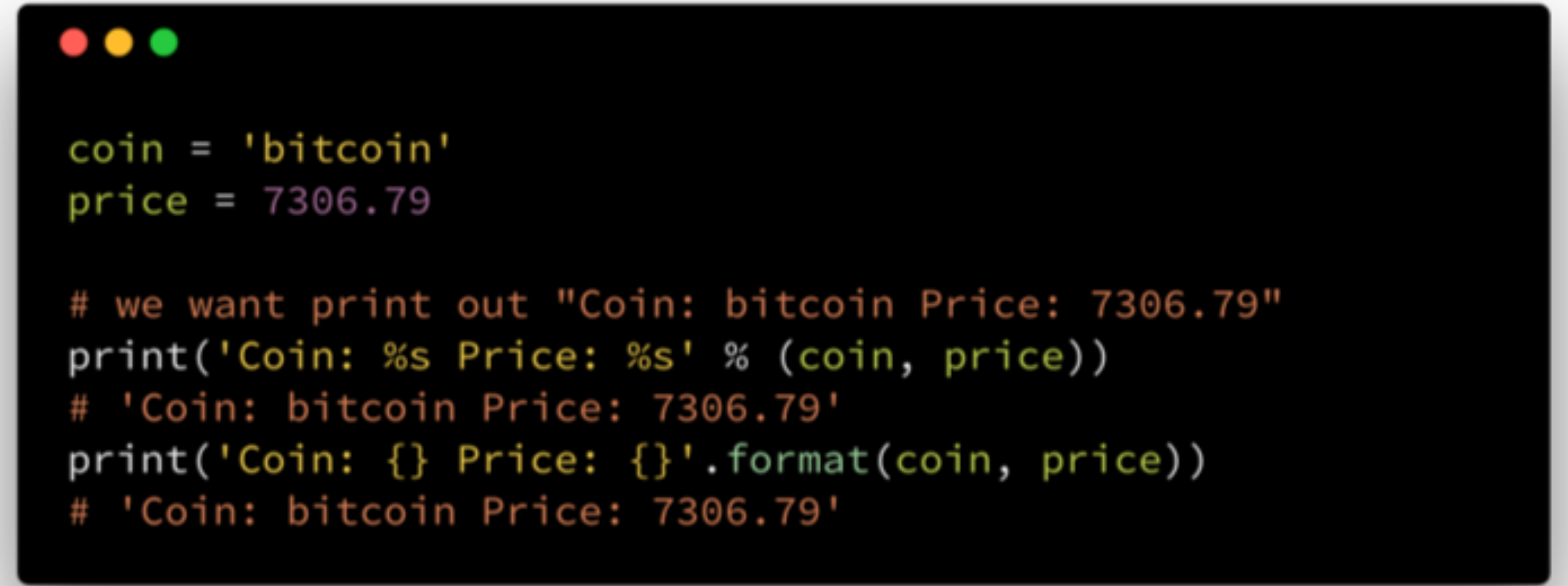
PEP 498 -- Literal String Interpolation (Python 3.6)

https://www.python.org/dev/peps/pep-0498/

String Formatting

```
coin = 'bitcoin'
price = 7306.79

# we want print out "Coin: bitcoin Price: 7306.79"
```

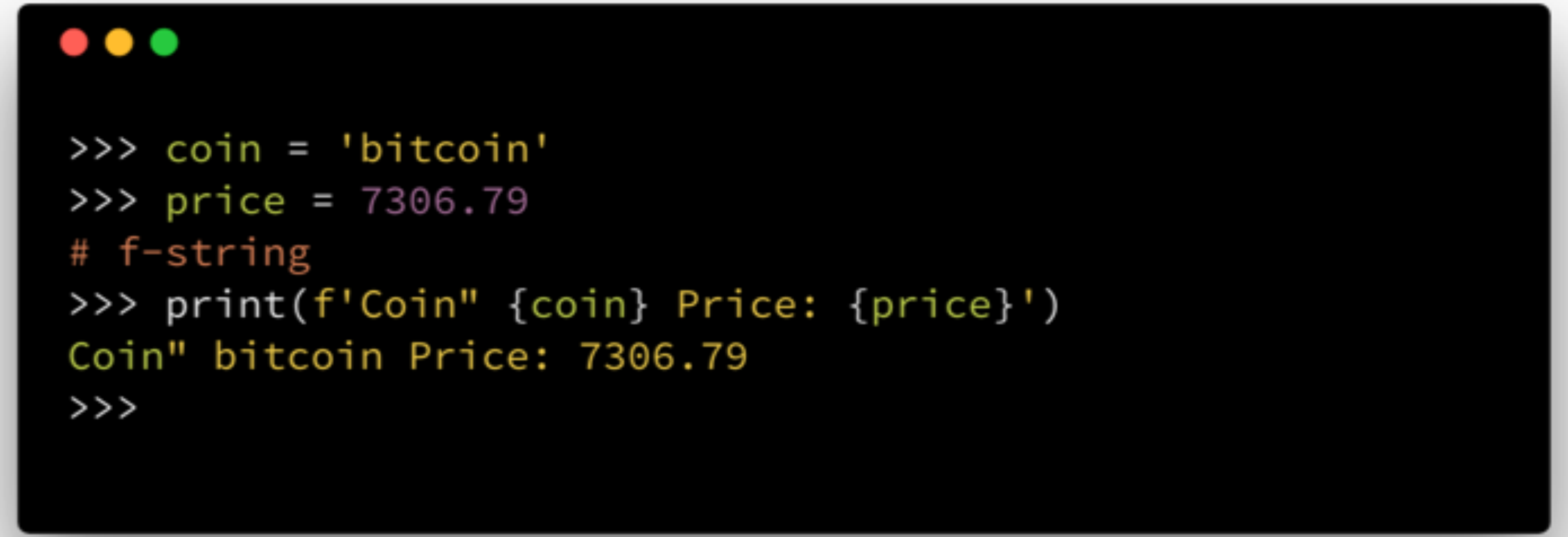
String Formatting



```
coin = 'bitcoin'
price = 7306.79

# we want print out "Coin: bitcoin Price: 7306.79"
print('Coin: %s Price: %s' % (coin, price))
# 'Coin: bitcoin Price: 7306.79'
print('Coin: {} Price: {}'.format(coin, price))
# 'Coin: bitcoin Price: 7306.79'
```

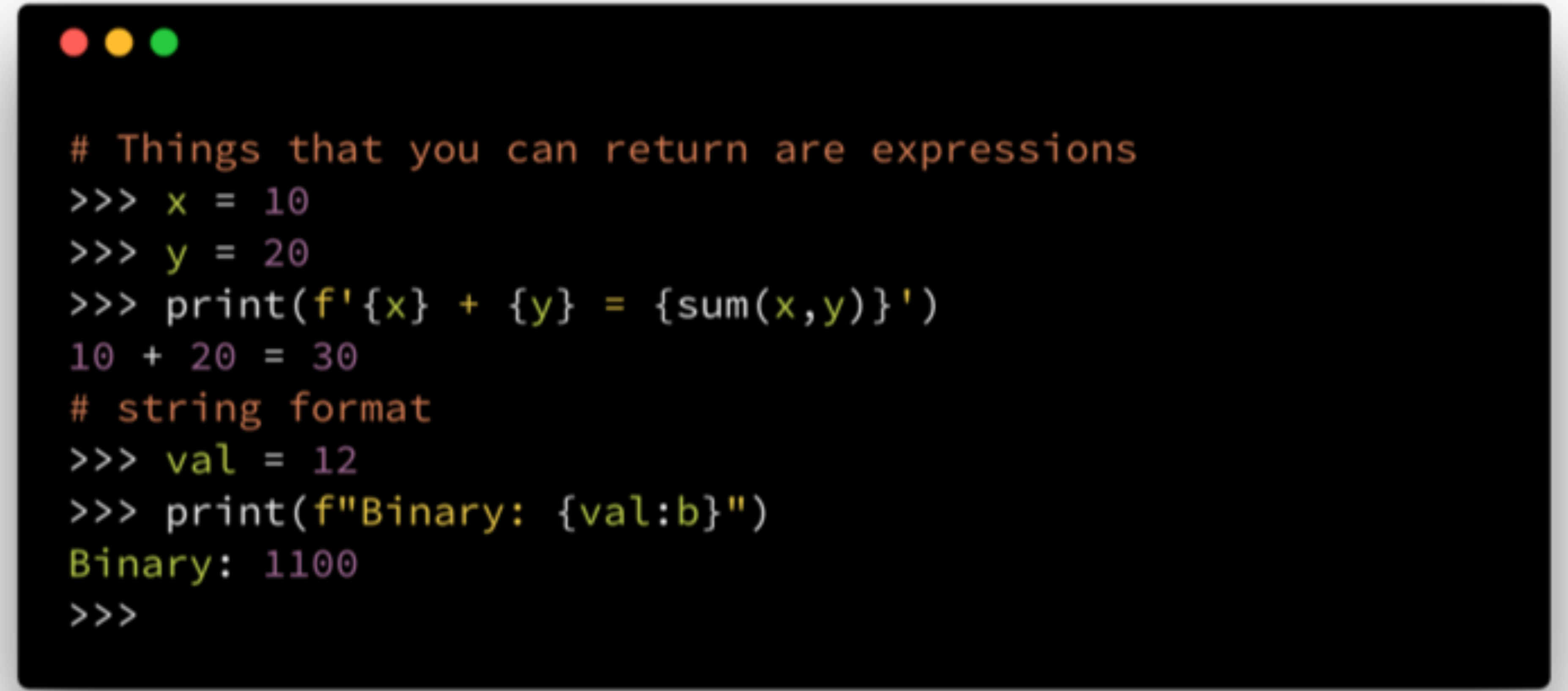
f-string

- If you precede a string literal with an **f** you can put an expression inside the {}



```
>>> coin = 'bitcoin'
>>> price = 7306.79
# f-string
>>> print(f'Coin" {coin} Price: {price}')
Coin" bitcoin Price: 7306.79
>>>
```

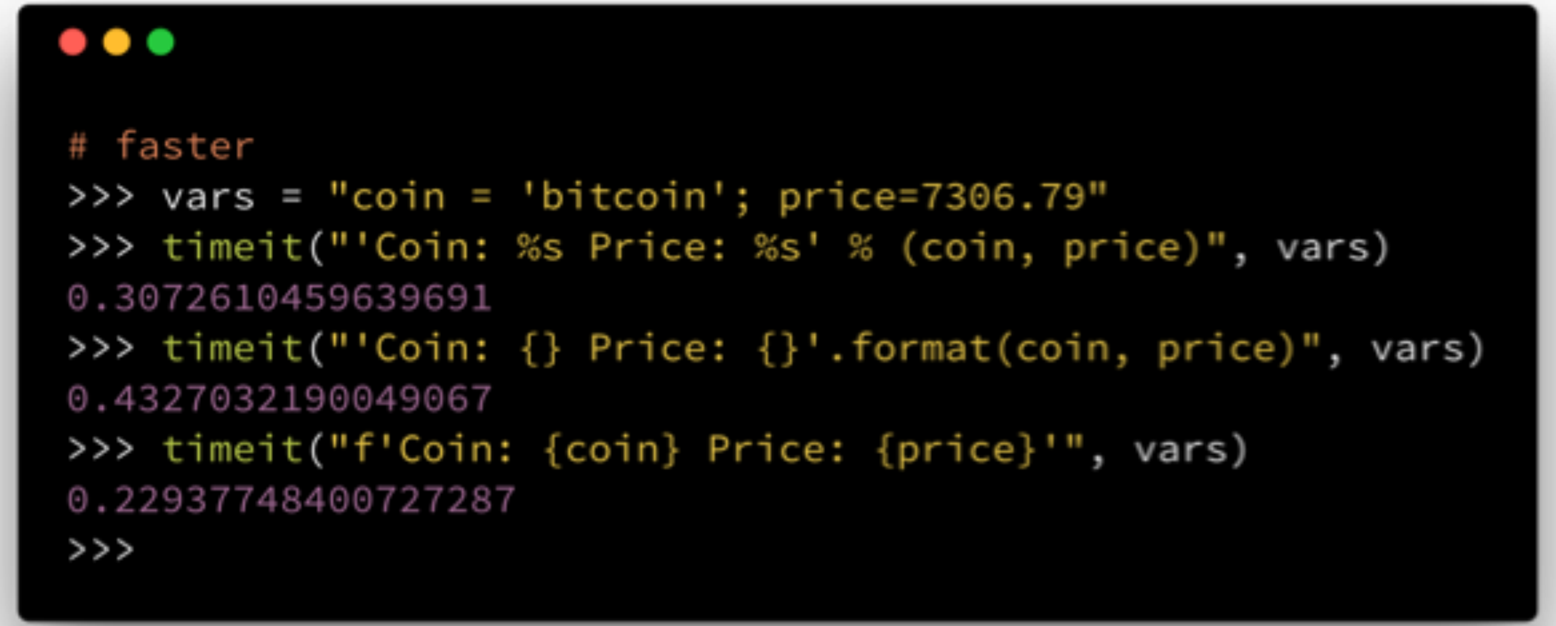
f-string: more examples



```
# Things that you can return are expressions
>>> x = 10
>>> y = 20
>>> print(f'{x} + {y} = {sum(x,y)}')
10 + 20 = 30
# string format
>>> val = 12
>>> print(f"Binary: {val:b}")
Binary: 1100
>>>
```

Why we should use f-string?

- Our code is more clear.
- Also flaster!



```
# faster
>>> vars = "coin = 'bitcoin'; price=7306.79"
>>> timeit("'Coin: %s Price: %s' % (coin, price)", vars)
0.3072610459639691
>>> timeit("'Coin: {} Price: {}'.format(coin, price)", vars)
0.4327032190049067
>>> timeit("f'Coin: {coin} Price: {price}'", vars)
0.22937748400727287
>>>
```

PEP 450 -- Adding A Statistics Module To The Standard Library (>=python 3.4)

https://www.python.org/dev/peps/pep-0450/

Mean

- mean is the average.

$$\mu = \frac{\sum X}{N}$$

Variance

- variance is the average squared deviation from the mean of a set of data, it is used to find the standard deviation

$$\frac{\sum (x-\mu)^2}{n}$$

Standard Deviation

- Standard Deviation shows the **variation** in data. If the data is close together, the standard deviation will be small. If the data is spread out, the standard deviation will be large.

$$\sigma = \sqrt{\frac{\sum (x-\mu)^2}{n}}$$

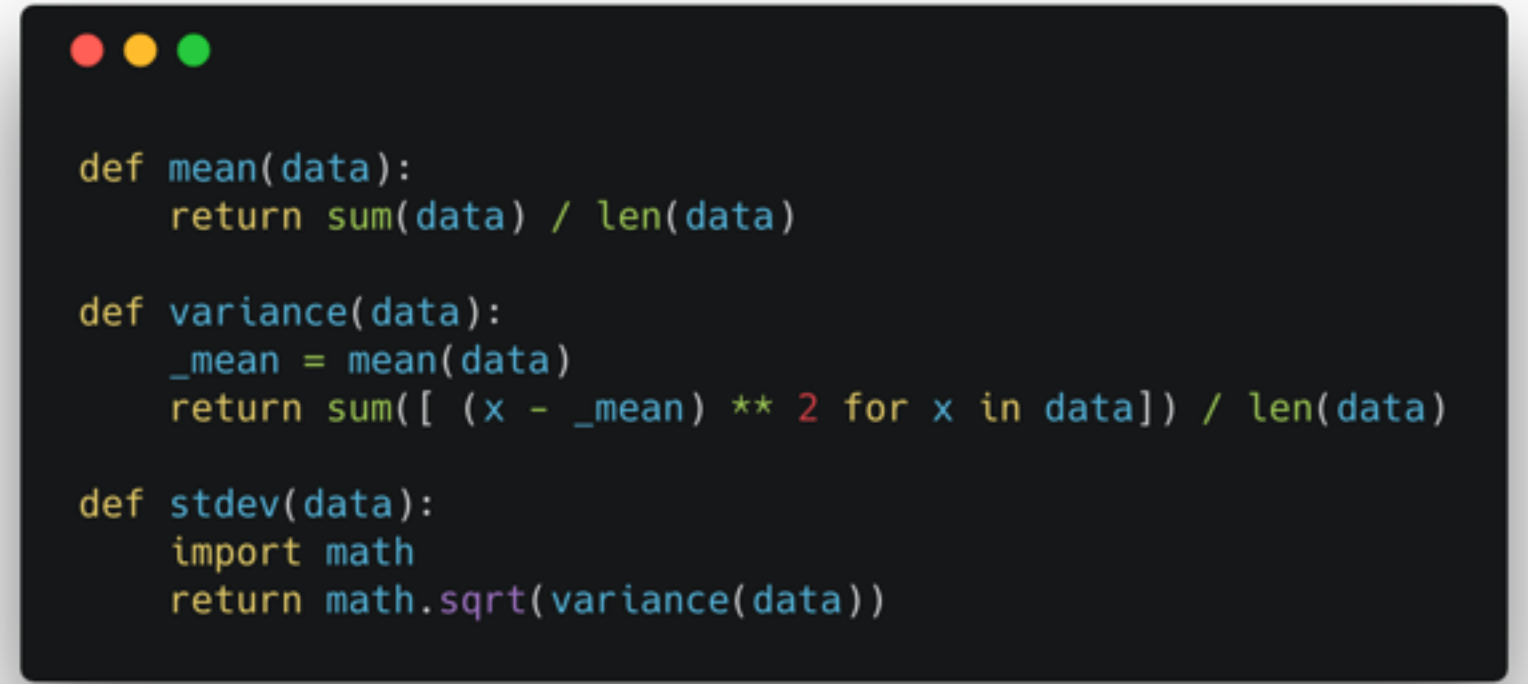
Example

Students test scores: 92, 88, 80, 68 and 52

- Find the **mean**: (92 + 88 + 80 + 68 + 52)/5 = 76
- Find the **variance** from the mean:
  - 92-76 = 16 - > square = 256
  - 88-76 = 12 - > square = 144
  - 80-76 = 4 - > square = 16
  - 68-76 = -8 - > square = 64
  - 52-76 = -24 - > square = 576

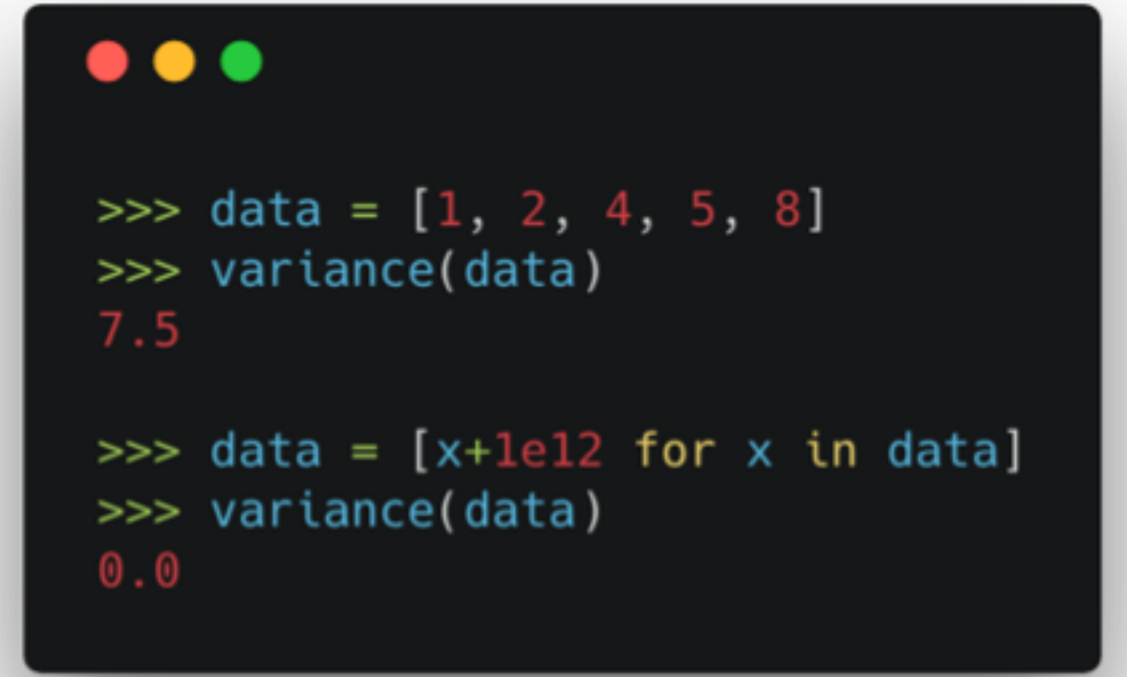
  sum = 1056 - > devide by 5 = 1056/5 =211.2
- Find the **standard deviation**: square root of deviation = $\sqrt{211.2}$ = 14.53

DIY Statistics Functions



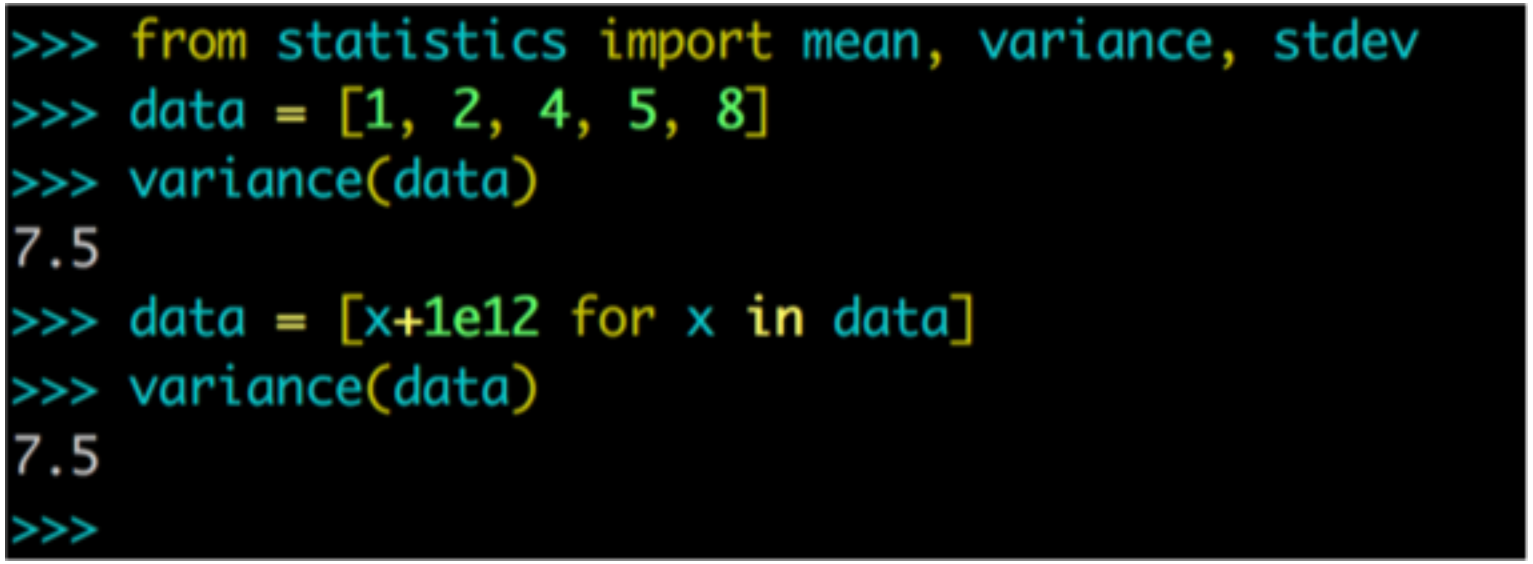
```python
def mean(data):
    return sum(data) / len(data)

def variance(data):
    _mean = mean(data)
    return sum([ (x - _mean) ** 2 for x in data]) / len(data)

def stdev(data):
    import math
    return math.sqrt(variance(data))
```

The problem



```
>>> data = [1, 2, 4, 5, 8]
>>> variance(data)
7.5

>>> data = [x+1e12 for x in data]
>>> variance(data)
0.0
```

Solution: Standard Statistic Module



```
>>> from statistics import mean, variance, stdev
>>> data = [1, 2, 4, 5, 8]
>>> variance(data)
7.5
>>> data = [x+1e12 for x in data]
>>> variance(data)
7.5
>>>
```

https://github.com/python/cpython/blob/master/Lib/statistics.py

PEP 515 -- Underscores in Numeric Literals (python 3.6)

https://www.python.org/dev/peps/pep-0515/

Syntax

```
# python 3.6
>>> a = 100_000_000
>>> a
100000000
>>> type(a)
<class 'int'>
```

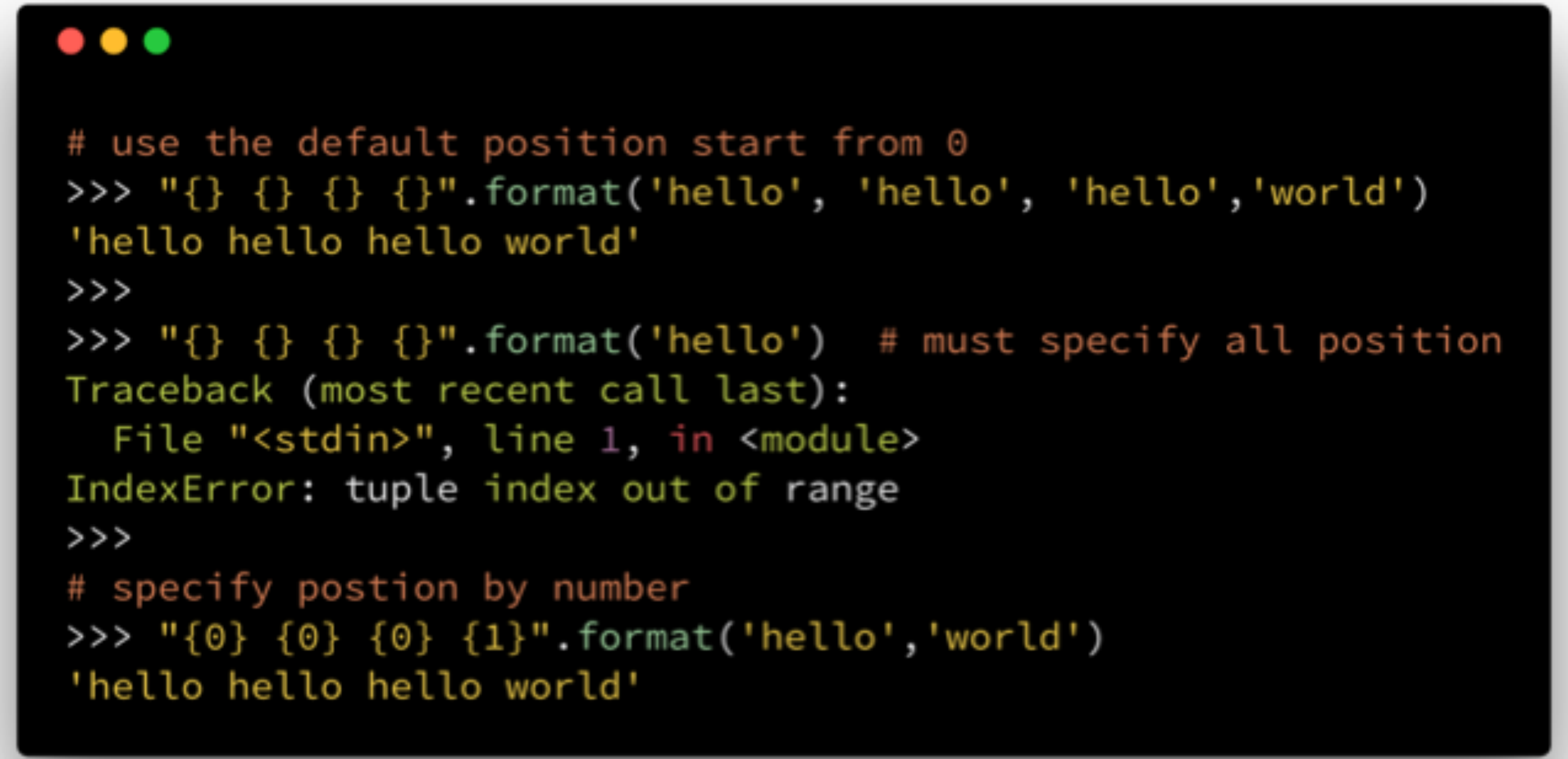
```
# python 2.7
>>> a = 1_2
  File "<stdin>", line 1
    a = 1_2
          ^
SyntaxError: invalid syntax
>>>
```

PEP 3101 -- **Advanced** String Formatting（python 3.0）

https://www.python.org/dev/peps/pep-3101/

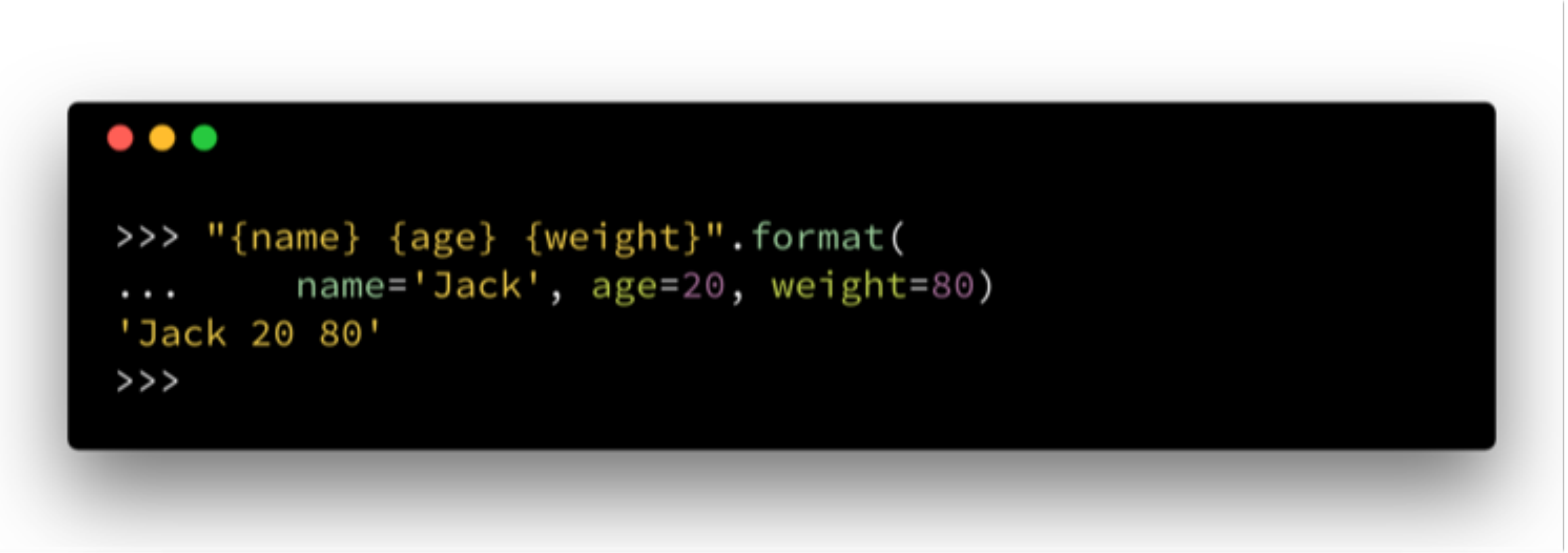
Basic Formatting

Specify by Position



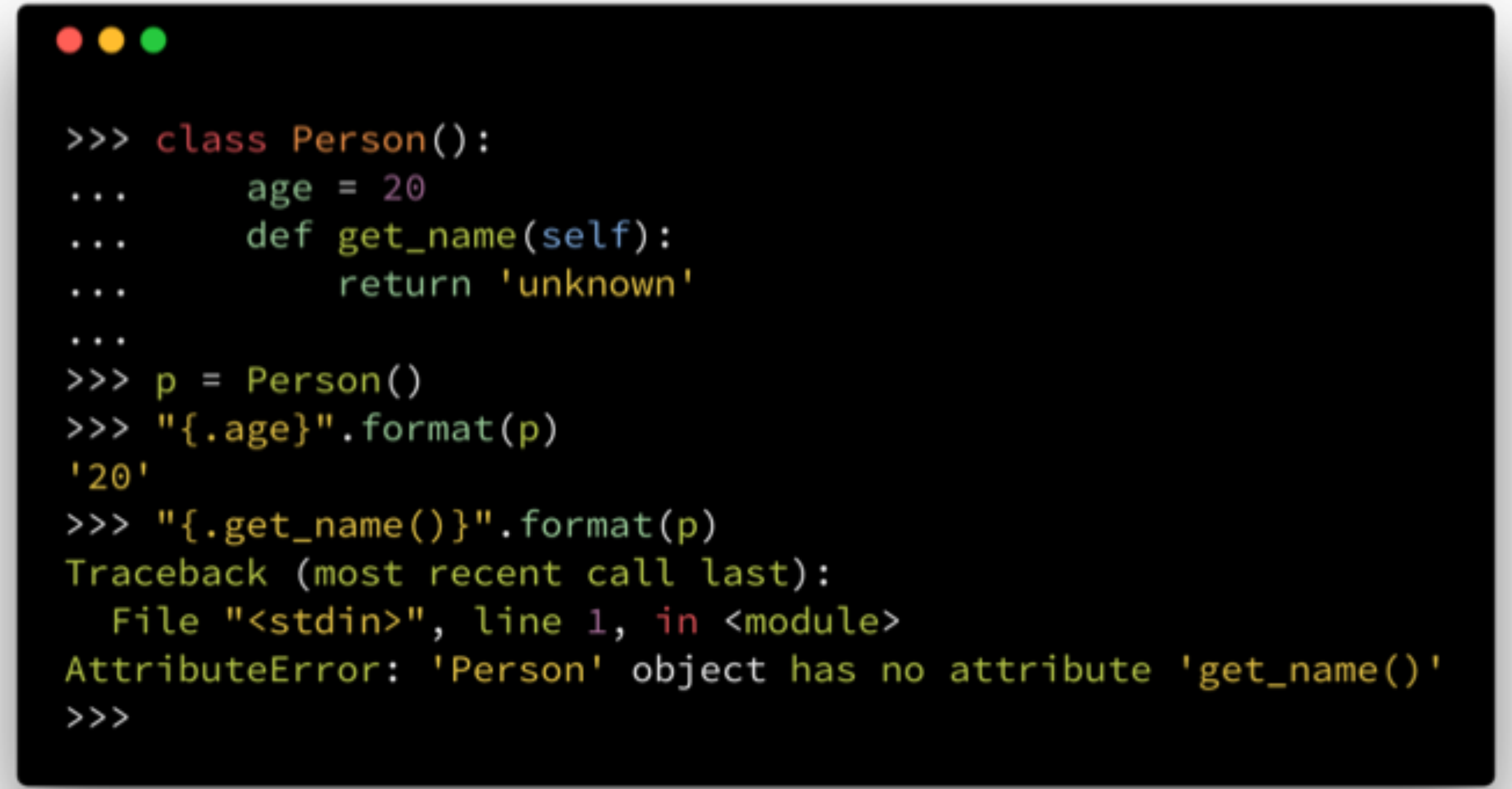
```
# use the default position start from 0
>>> "{} {} {} {}".format('hello', 'hello', 'hello','world')
'hello hello hello world'
>>>
>>> "{} {} {} {}".format('hello')  # must specify all position
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
>>>
# specify postion by number
>>> "{0} {0} {0} {1}".format('hello','world')
'hello hello hello world'
```

Specify by Keyword



```
>>> "{name} {age} {weight}".format(
...     name='Jack', age=20, weight=80)
'Jack 20 80'
>>>
```

Access attibutes not methods

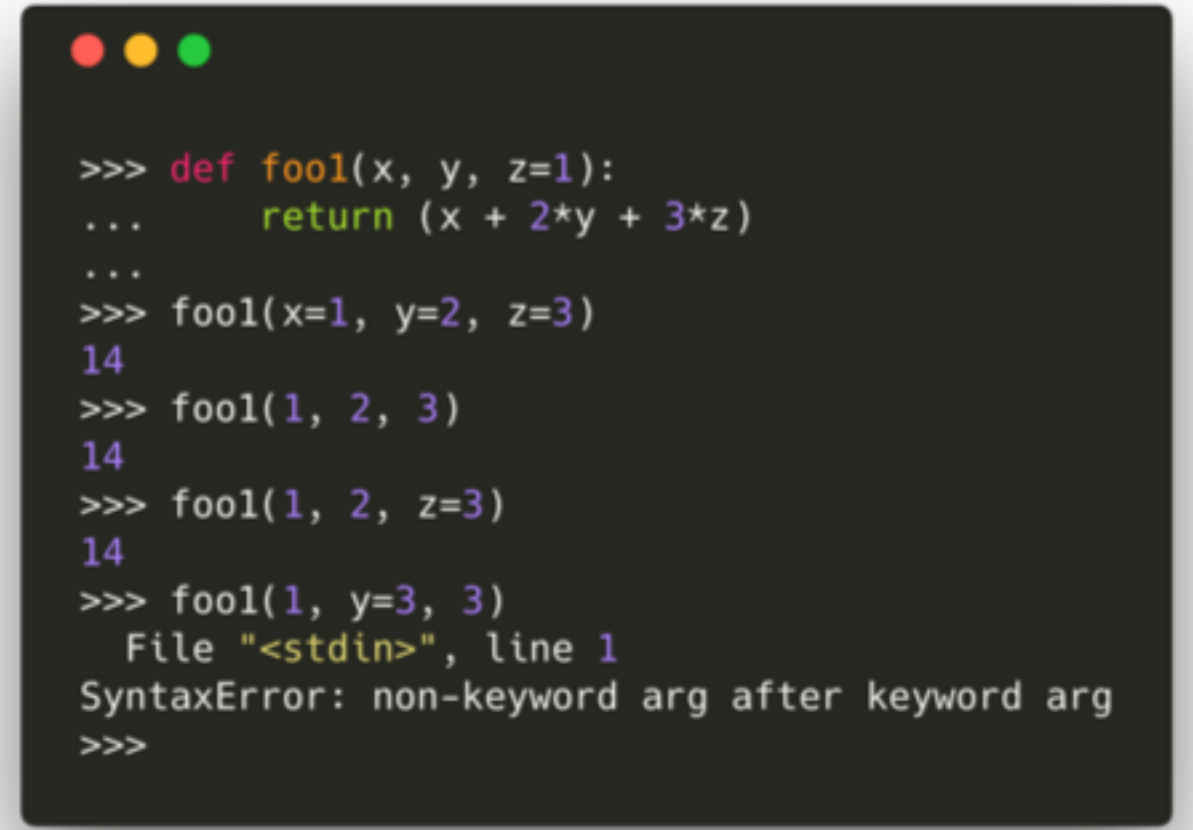


```
>>> class Person():
...     age = 20
...     def get_name(self):
...         return 'unknown'
...
>>> p = Person()
>>> "{.age}".format(p)
'20'
>>> "{.get_name()}".format(p)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Person' object has no attribute 'get_name()'
>>>
```

PEP 3102 -- Keyword-Only Arguments (>=python 3.0)

https://www.python.org/dev/peps/pep-3102/

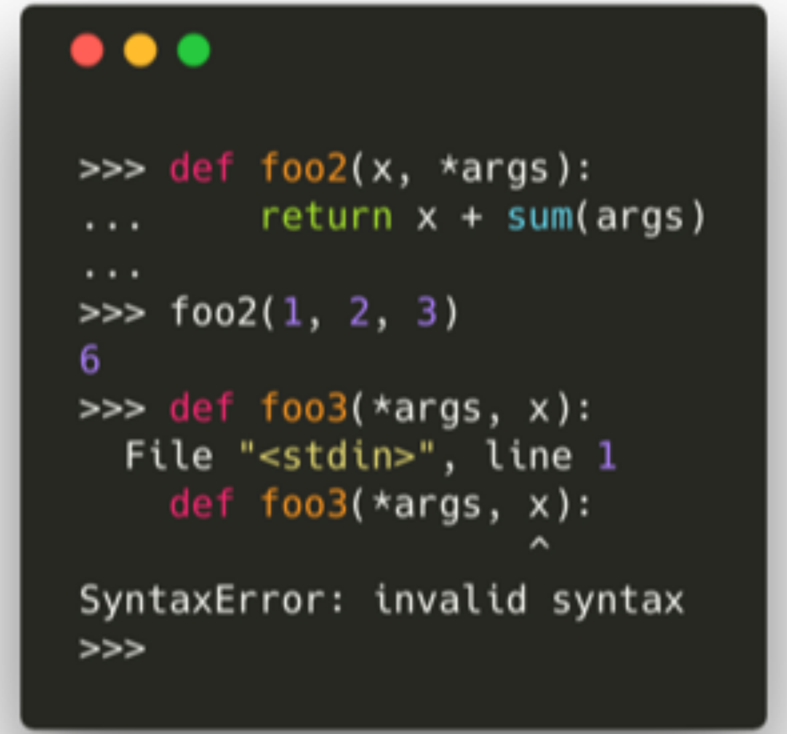
Position or Keyword Argument



```
>>> def foo1(x, y, z=1):
...     return (x + 2*y + 3*z)
...
>>> foo1(x=1, y=2, z=3)
14
>>> foo1(1, 2, 3)
14
>>> foo1(1, 2, z=3)
14
>>> foo1(1, y=3, 3)
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
>>>
```

Variable number of arguments

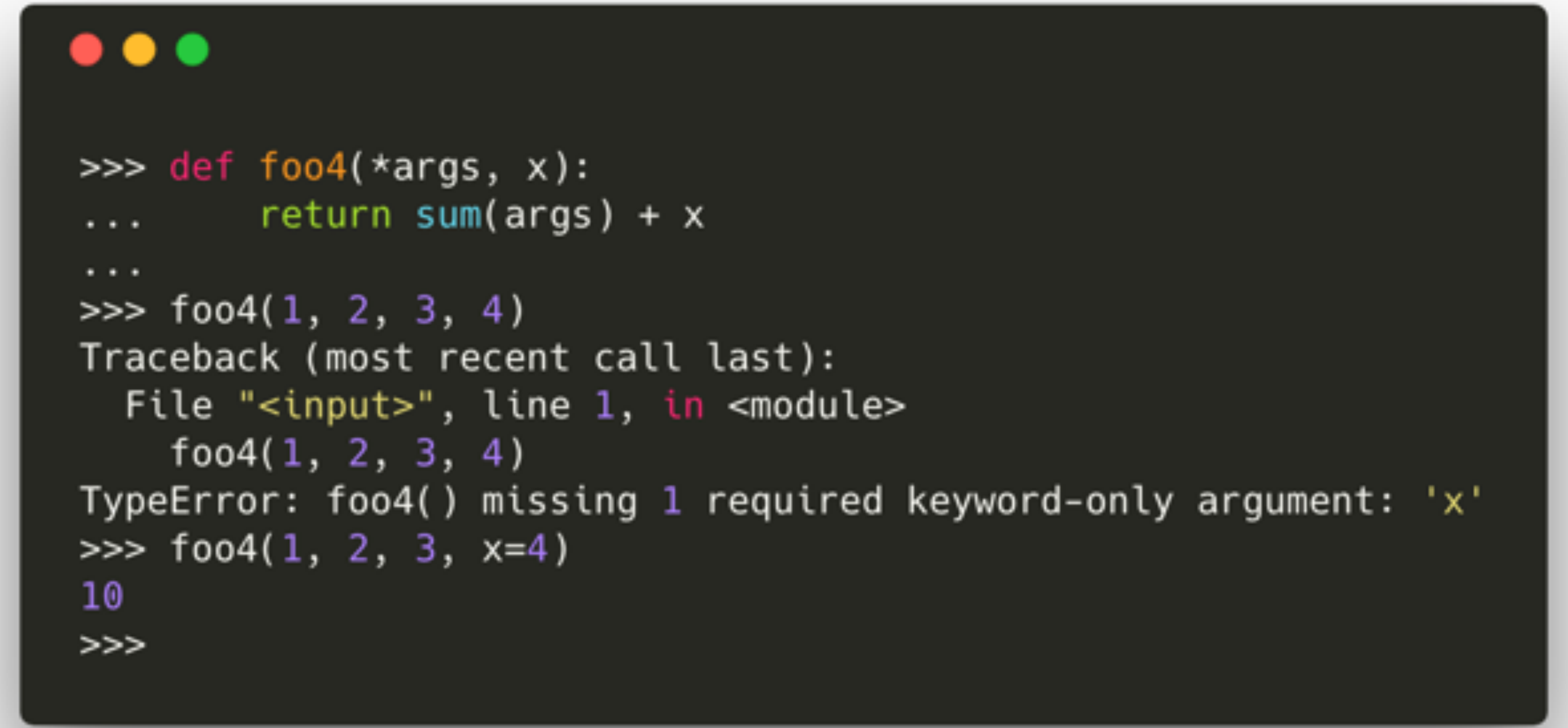
- There are often cases where it is desirable for a function to take **a variable number of arguments**. The Python language supports this using the 'varargs' syntax (*name), which specifies that any 'left over' arguments be passed into the varargs parameter **as a tuple**.
- One limitation on this is that currently for Python 2, named arguments can not be after varargs.



```
>>> def foo2(x, *args):
...     return x + sum(args)
...
>>> foo2(1, 2, 3)
6
>>> def foo3(*args, x):
  File "<stdin>", line 1
    def foo3(*args, x):
                    ^
SyntaxError: invalid syntax
>>>
```

Python 3 Only

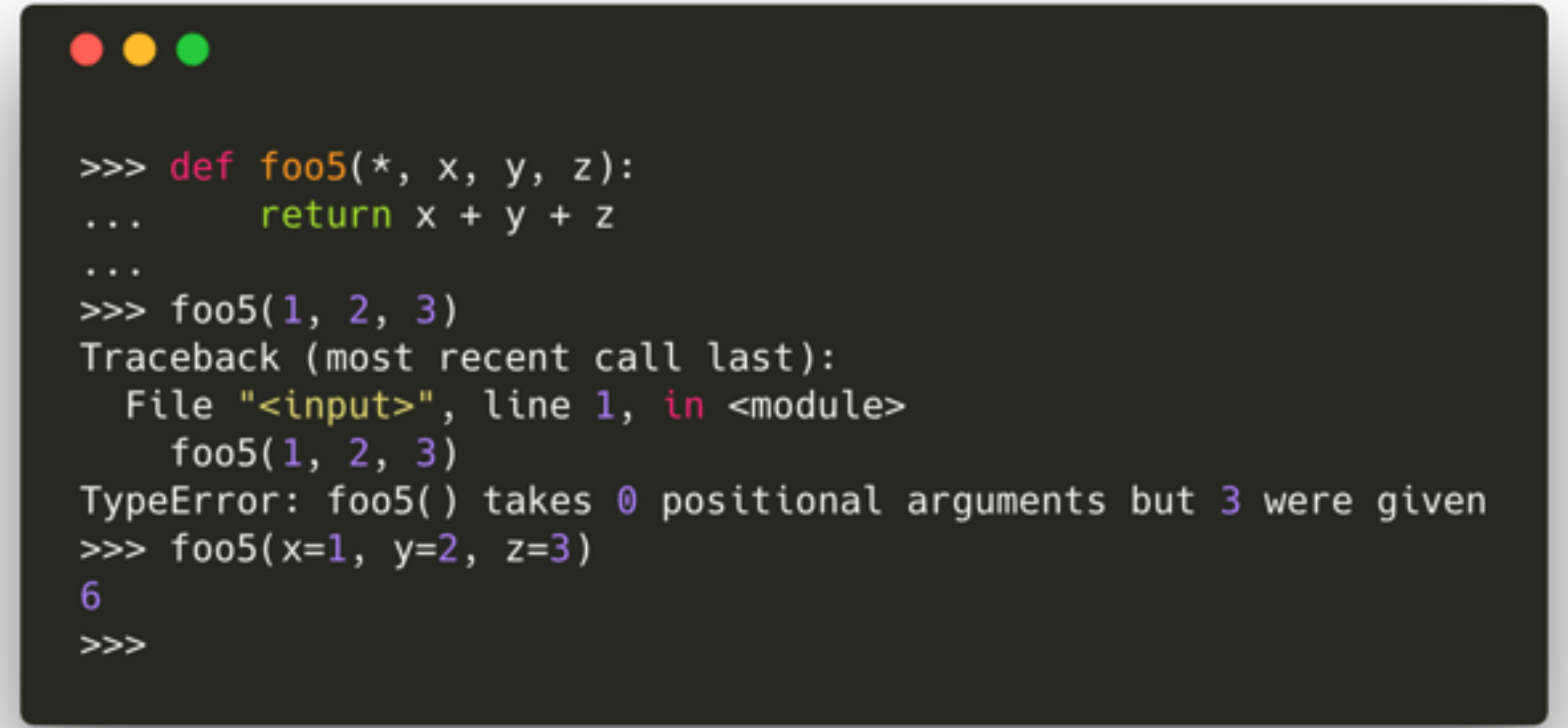
Named args after varargs



```
>>> def foo4(*args, x):
...     return sum(args) + x
...
>>> foo4(1, 2, 3, 4)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    foo4(1, 2, 3, 4)
TypeError: foo4() missing 1 required keyword-only argument: 'x'
>>> foo4(1, 2, 3, x=4)
10
>>>
```

Python 3 Only

Can use bare *



```
>>> def foo5(*, x, y, z):
...     return x + y + z
...
>>> foo5(1, 2, 3)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    foo5(1, 2, 3)
TypeError: foo5() takes 0 positional arguments but 3 were given
>>> foo5(x=1, y=2, z=3)
6
>>>
```

Keyword only

- Improves readability of functions by removing positional arguments.

send(404, 200, 100)   vs   **send**(code=404, amout=200, timeout=100)

PEP 3105 -- Make print a function

Syntax

```
# python 3
>>> print (value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

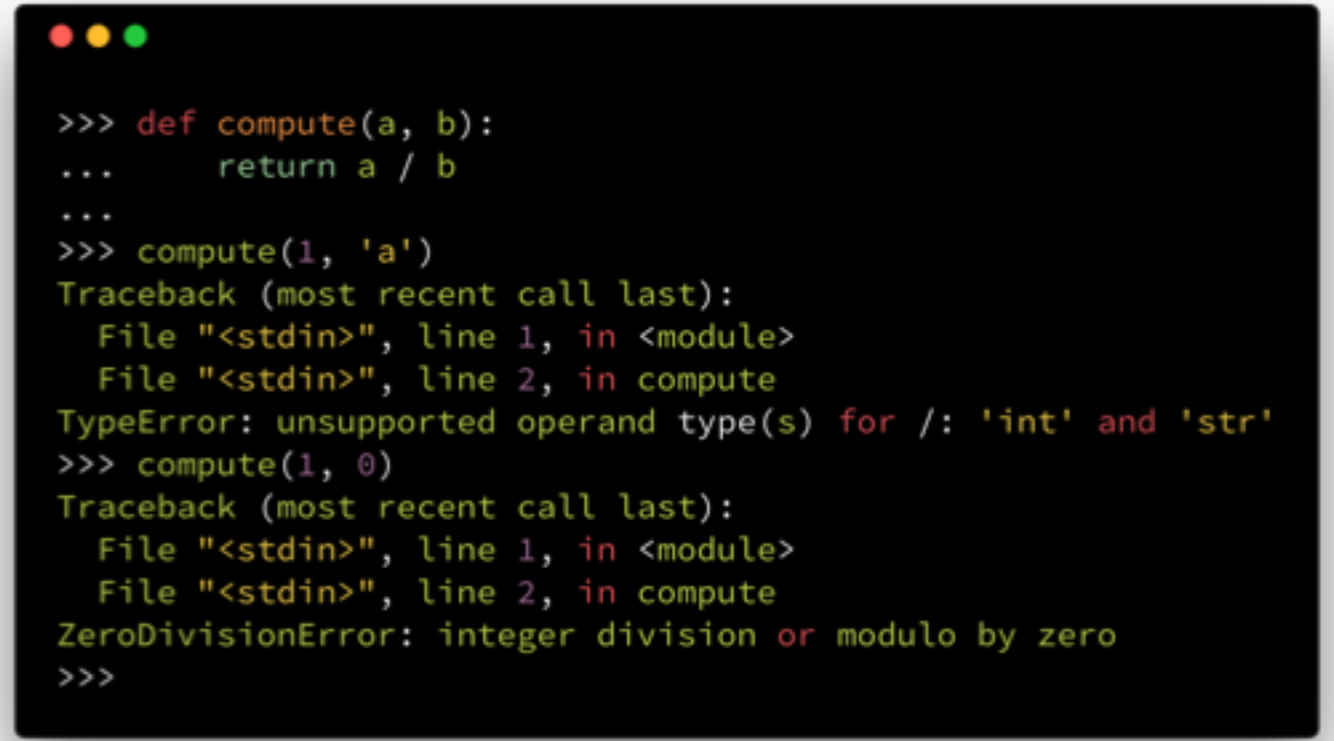
```
# python 2
>>> print  statement
```

Demo

PEP 3110 -- Catching Exceptions in Python 3000 (Python 3.0)

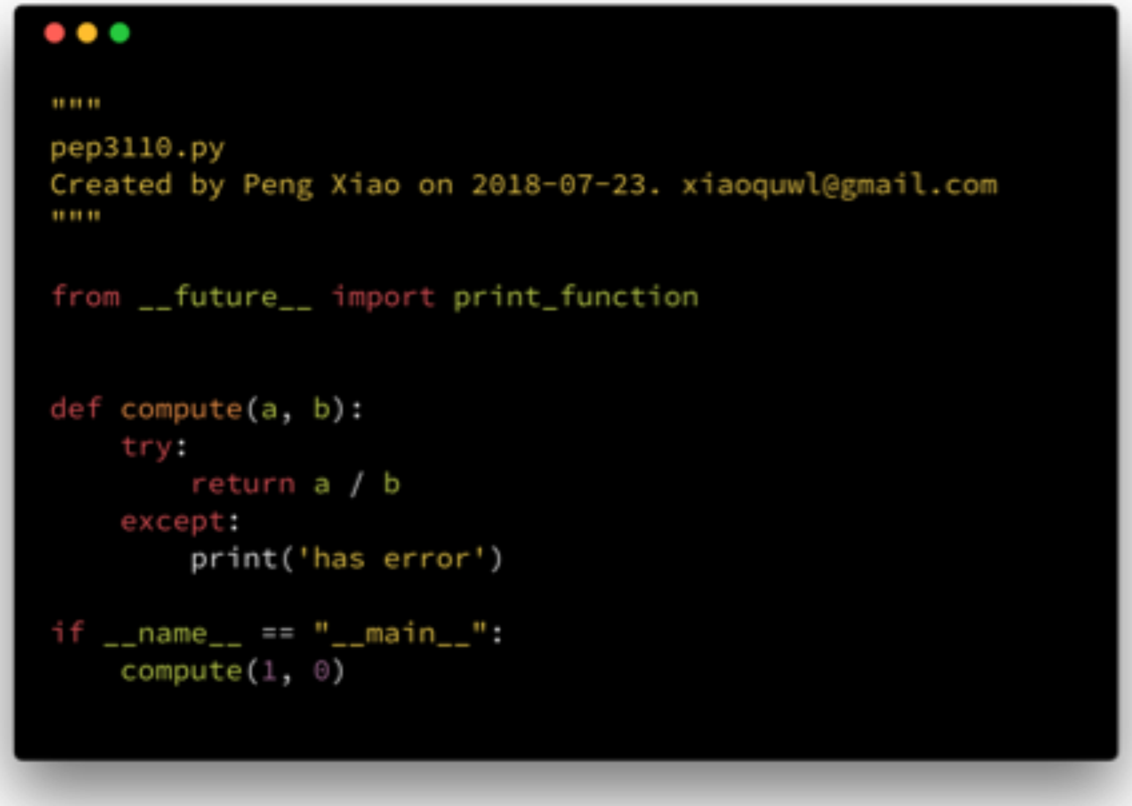
https://www.python.org/dev/peps/pep-3110/

Exceptions are important

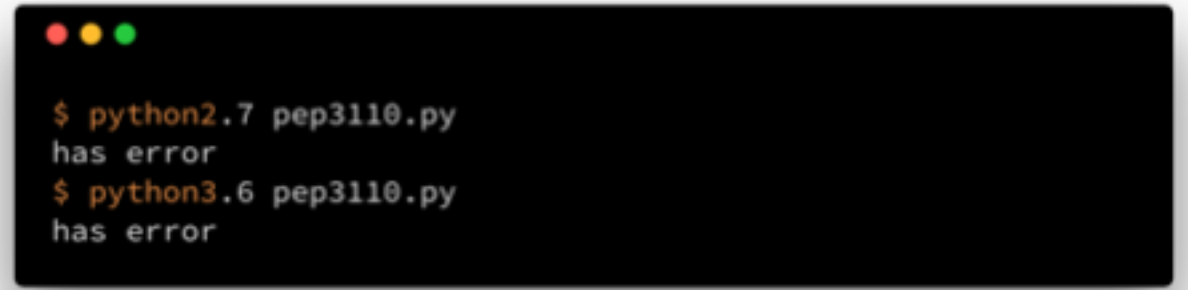


```
>>> def compute(a, b):
...     return a / b
...
>>> compute(1, 'a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in compute
TypeError: unsupported operand type(s) for /: 'int' and 'str'
>>> compute(1, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in compute
ZeroDivisionError: integer division or modulo by zero
>>>
```

Exceptions are important

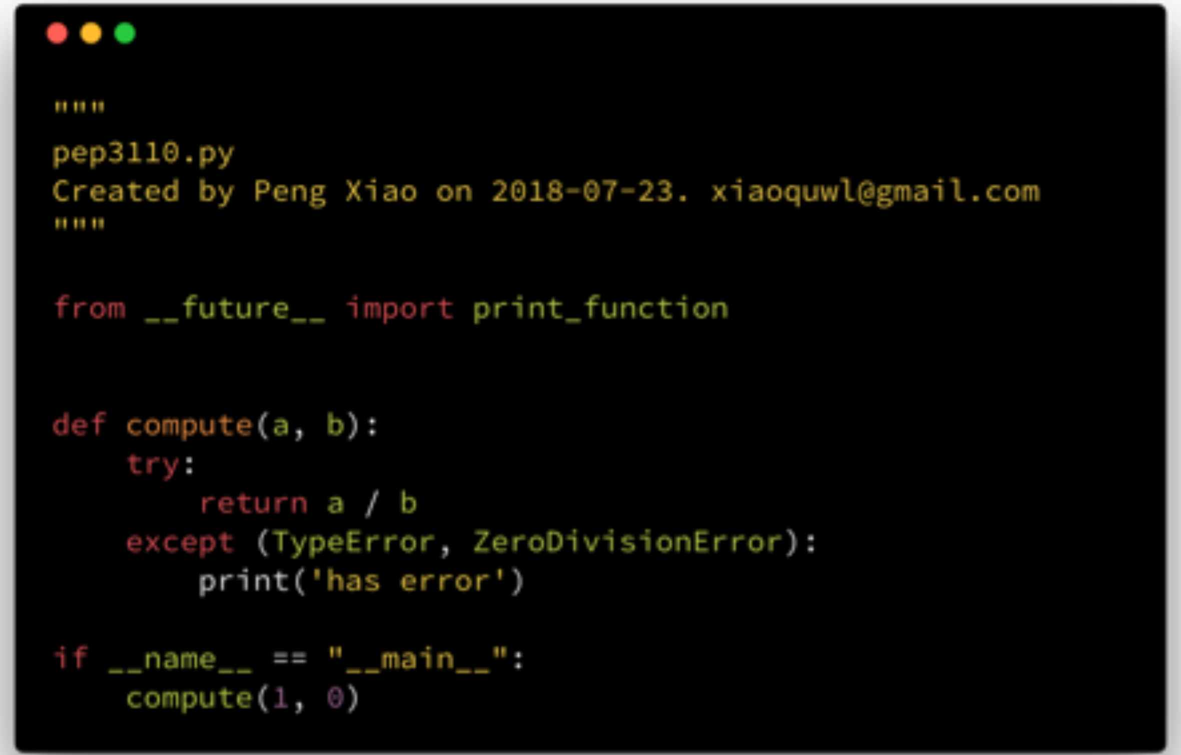


```
"""
pep3110.py
Created by Peng Xiao on 2018-07-23. xiaoquwl@gmail.com
"""

from __future__ import print_function


def compute(a, b):
    try:
        return a / b
    except:
        print('has error')

if __name__ == "__main__":
    compute(1, 0)
```

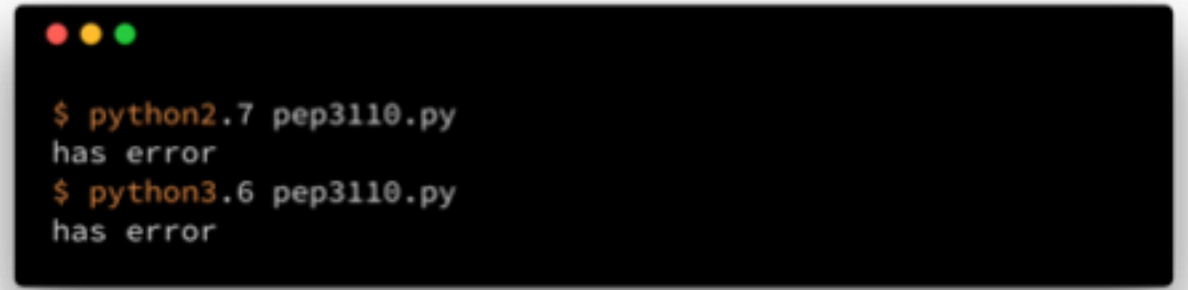


```
$ python2.7 pep3110.py
has error
$ python3.6 pep3110.py
has error
```

Exceptions are important

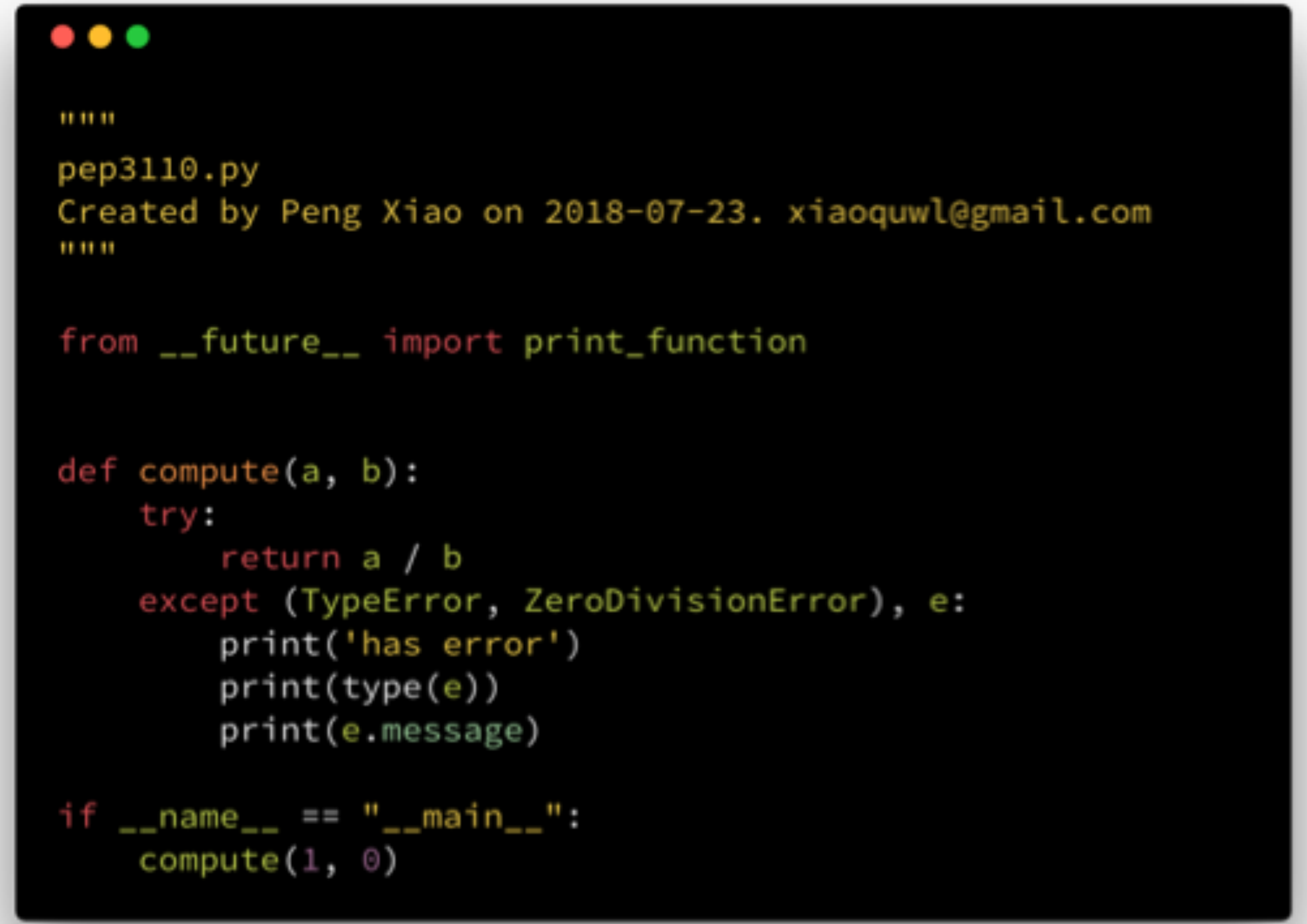


```python
"""
pep3110.py
Created by Peng Xiao on 2018-07-23. xiaoquwl@gmail.com
"""

from __future__ import print_function


def compute(a, b):
    try:
        return a / b
    except (TypeError, ZeroDivisionError):
        print('has error')

if __name__ == "__main__":
    compute(1, 0)
```



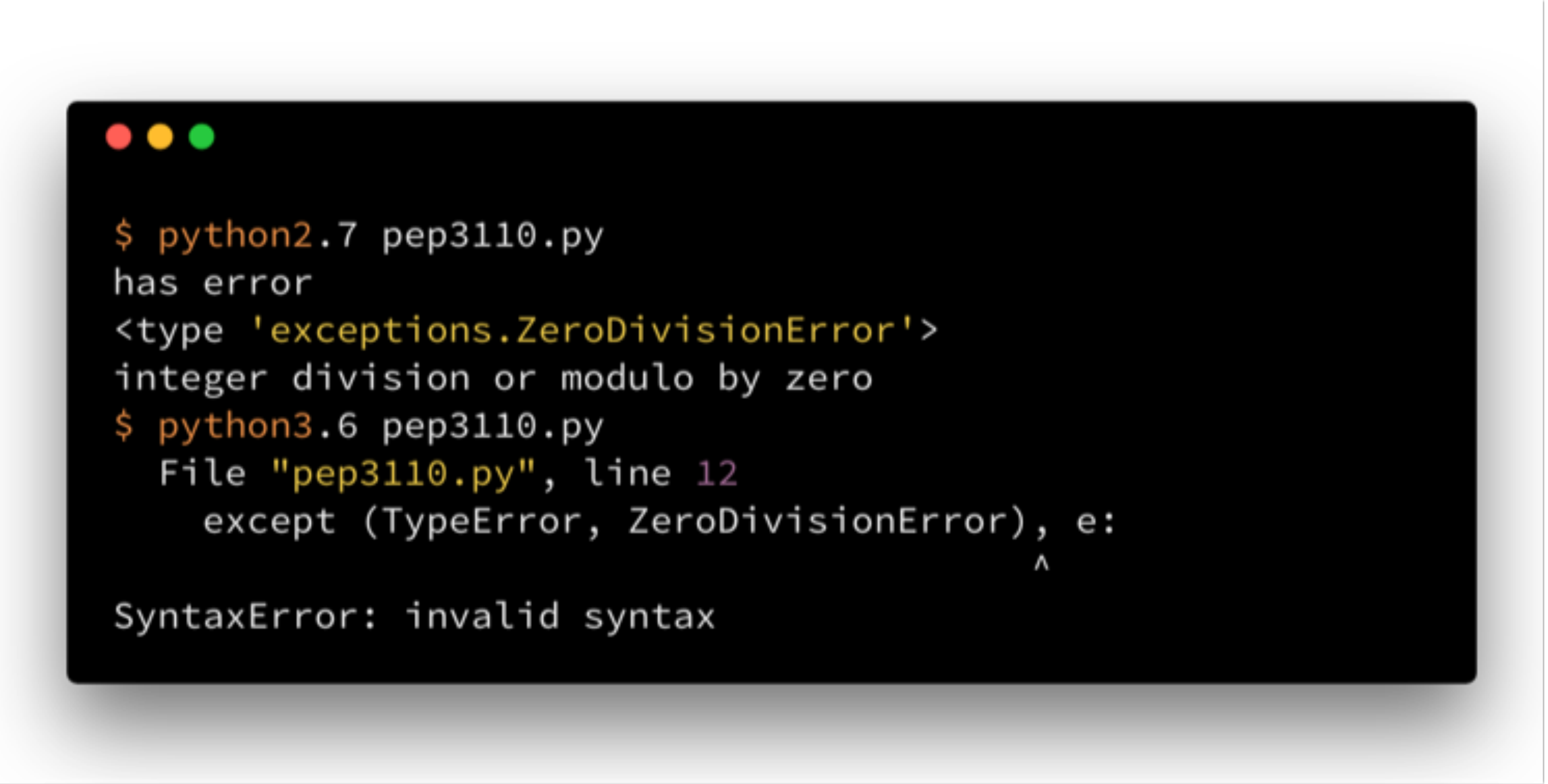
```
$ python2.7 pep3110.py
has error
$ python3.6 pep3110.py
has error
```

The Exception Type



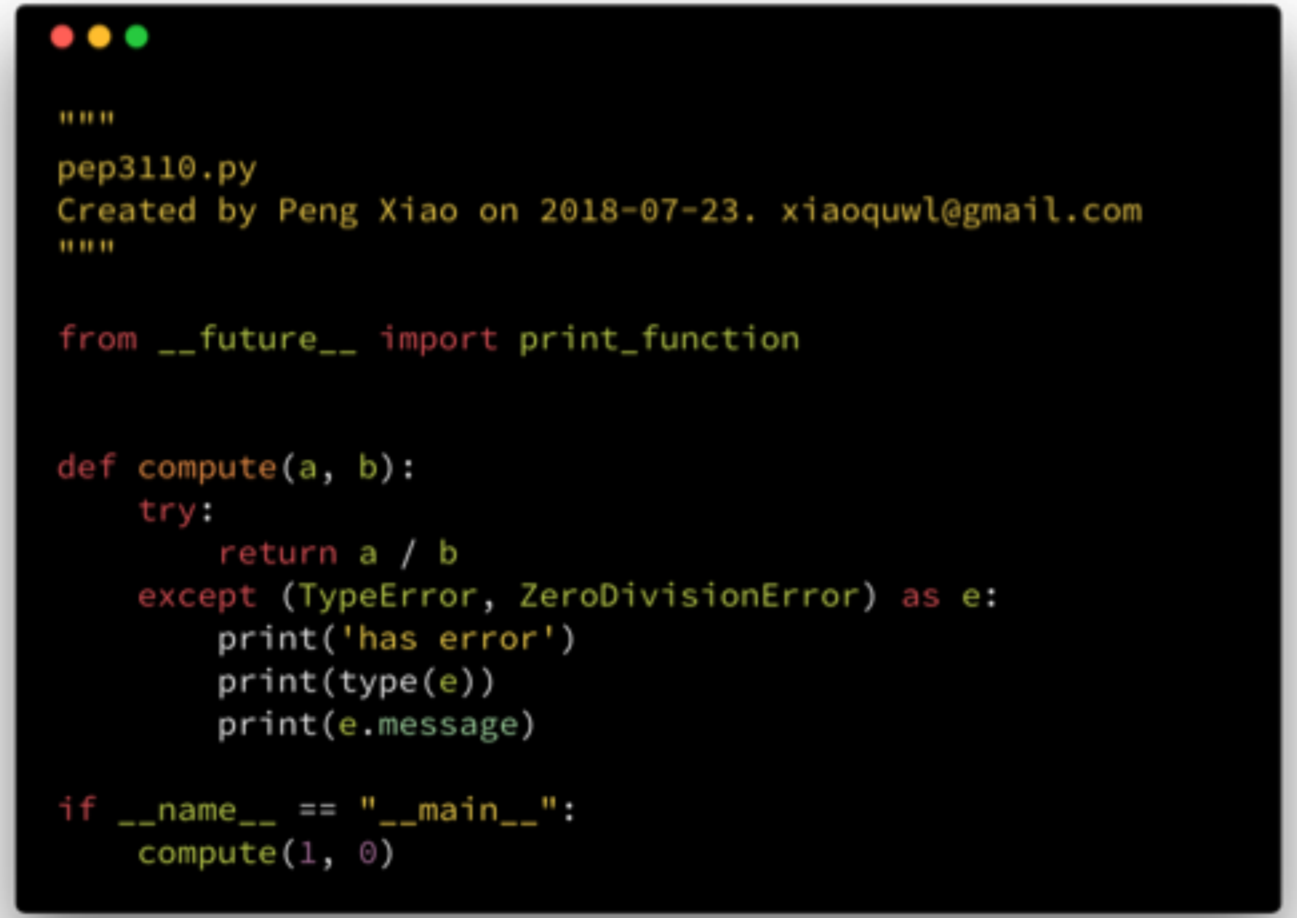
```
"""
pep3110.py
Created by Peng Xiao on 2018-07-23. xiaoquwl@gmail.com
"""

from __future__ import print_function


def compute(a, b):
    try:
        return a / b
    except (TypeError, ZeroDivisionError), e:
        print('has error')
        print(type(e))
        print(e.message)

if __name__ == "__main__":
    compute(1, 0)
```

The Exception Type



```
$ python2.7 pep3110.py
has error
<type 'exceptions.ZeroDivisionError'>
integer division or modulo by zero
$ python3.6 pep3110.py
  File "pep3110.py", line 12
    except (TypeError, ZeroDivisionError), e:
                                         ^
SyntaxError: invalid syntax
```

For Python 3

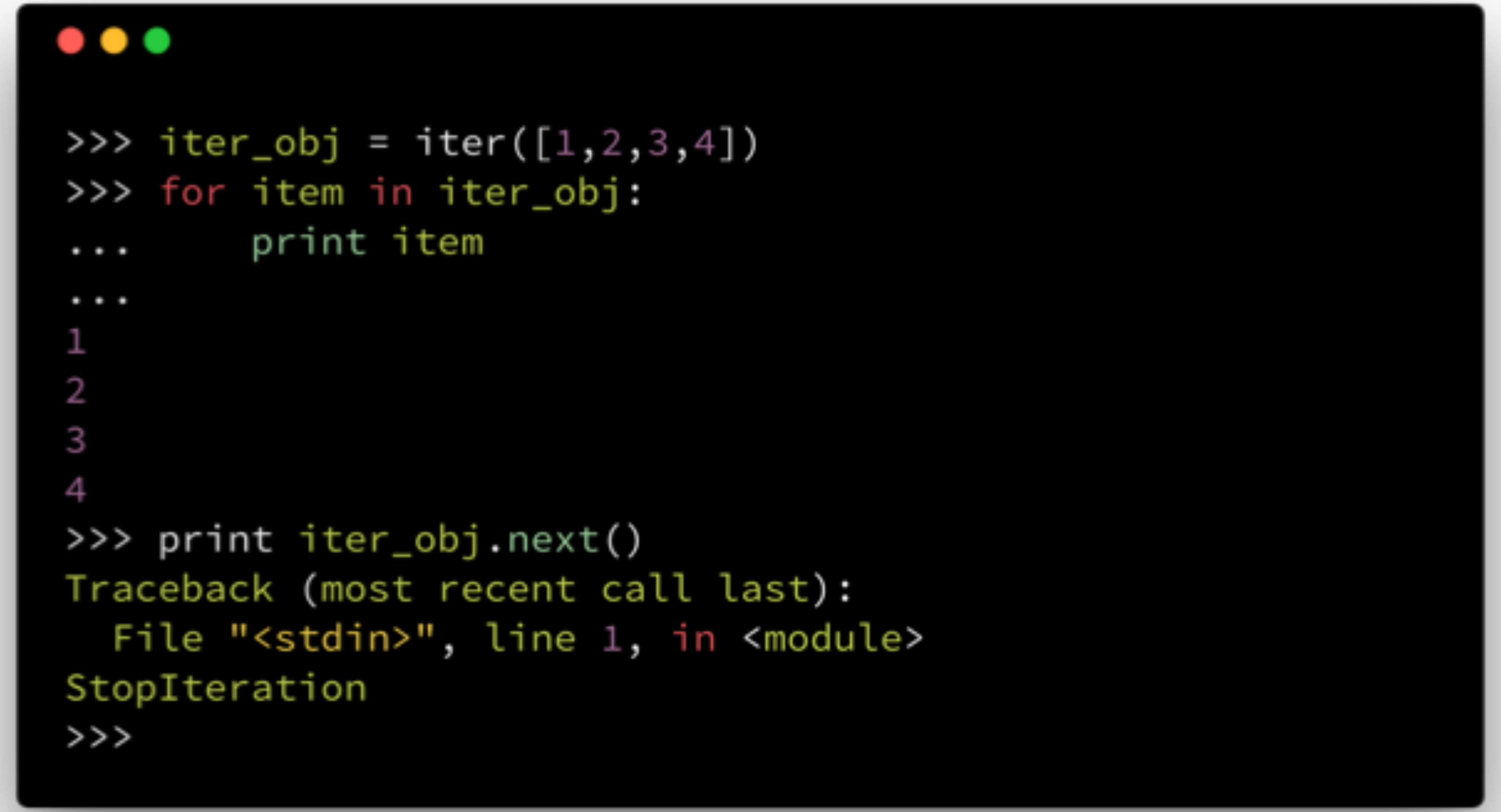


```
"""
pep3110.py
Created by Peng Xiao on 2018-07-23. xiaoquwl@gmail.com
"""

from __future__ import print_function


def compute(a, b):
    try:
        return a / b
    except (TypeError, ZeroDivisionError) as e:
        print('has error')
        print(type(e))
        print(e.message)

if __name__ == "__main__":
    compute(1, 0)
```

PEP 3114 -- Renaming iterator.next() to iterator.__next__() (Python 3.0)

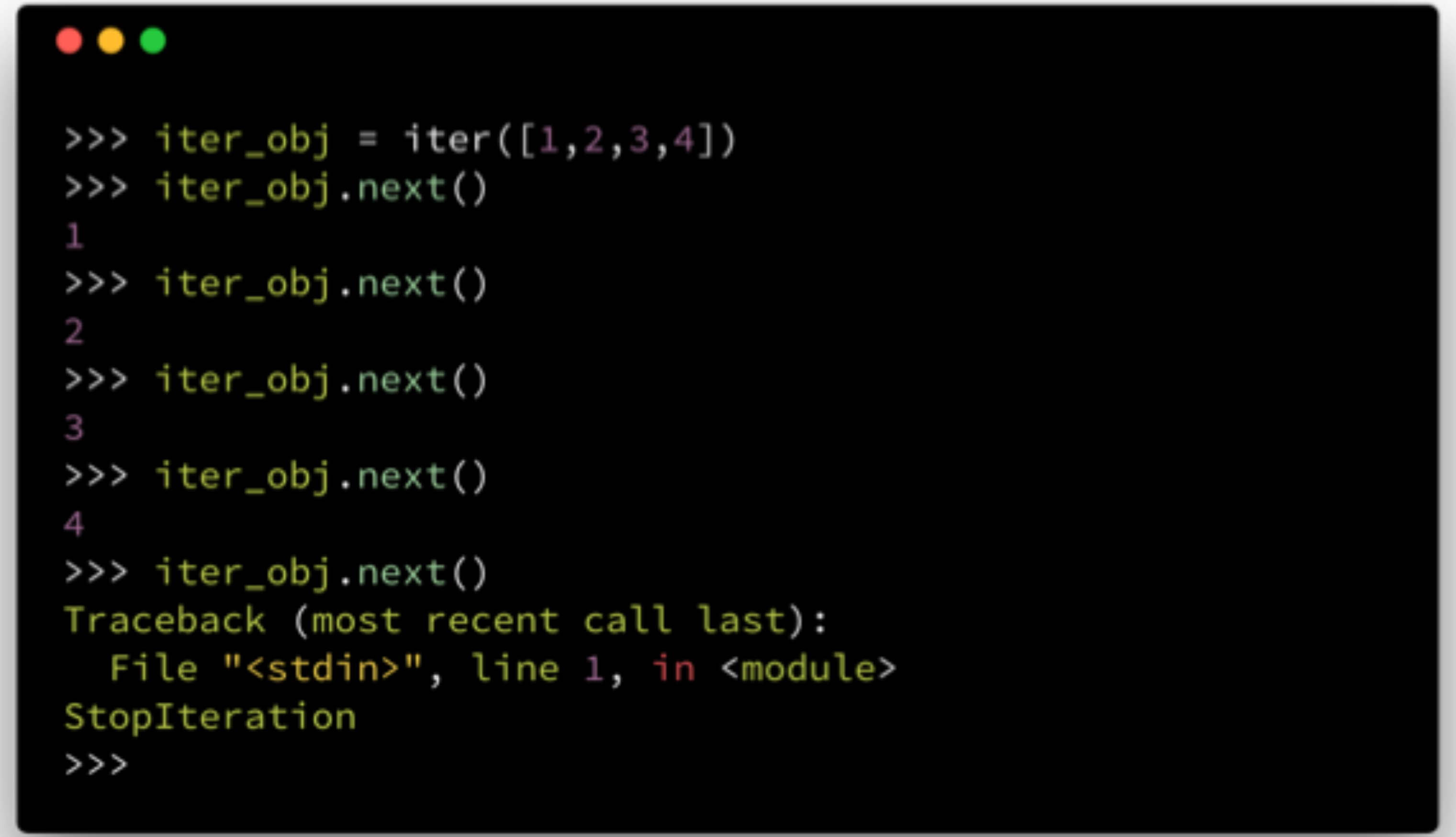
https://www.python.org/dev/peps/pep-3114/

Python 2 Iterator



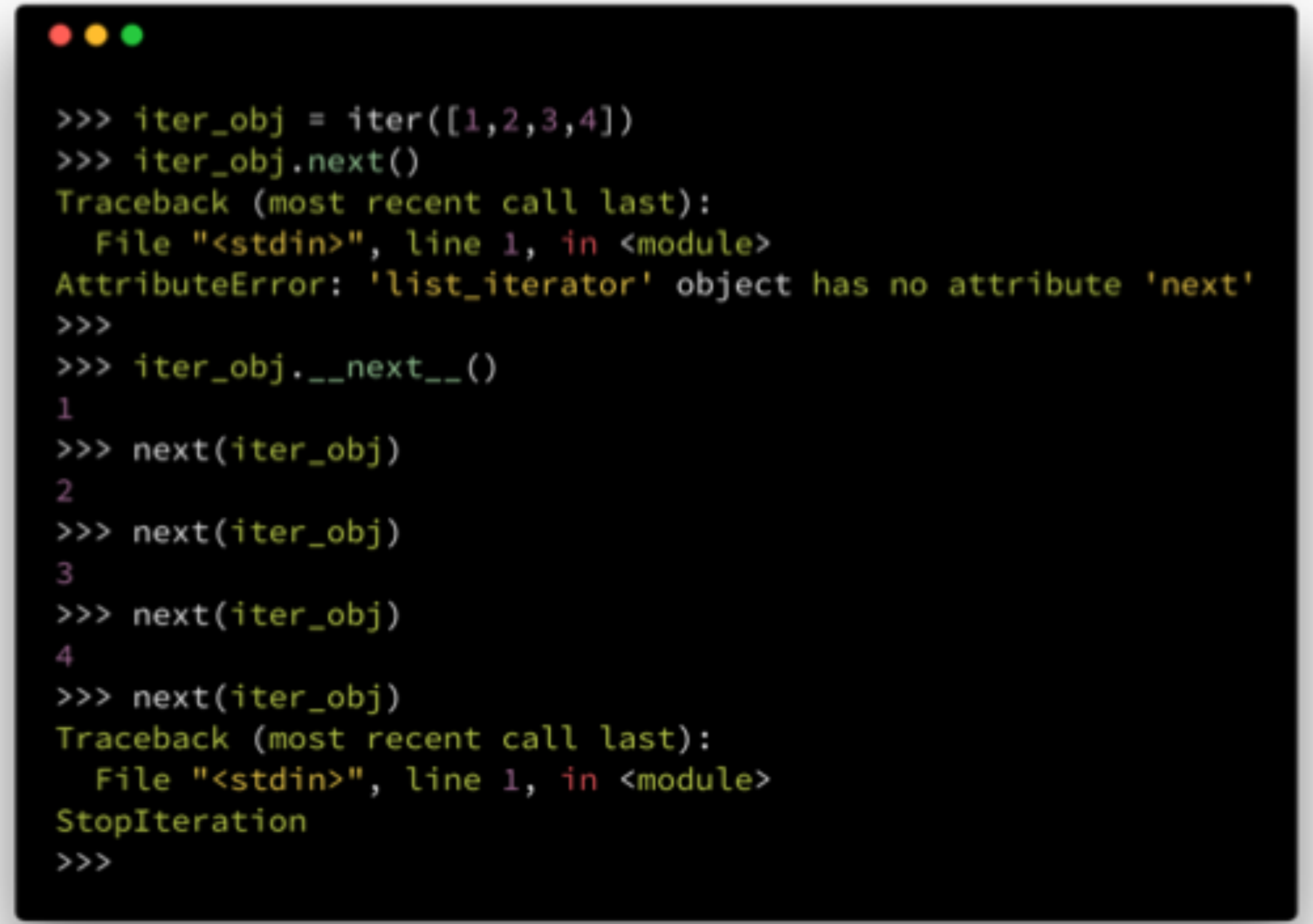
```
>>> iter_obj = iter([1,2,3,4])
>>> for item in iter_obj:
...     print item
...
1
2
3
4
>>> print iter_obj.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Python 2 Iterator



```
>>> iter_obj = iter([1,2,3,4])
>>> iter_obj.next()
1
>>> iter_obj.next()
2
>>> iter_obj.next()
3
>>> iter_obj.next()
4
>>> iter_obj.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Python 3 Iterator



```
>>> iter_obj = iter([1,2,3,4])
>>> iter_obj.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list_iterator' object has no attribute 'next'
>>>
>>> iter_obj.__next__()
1
>>> next(iter_obj)
2
>>> next(iter_obj)
3
>>> next(iter_obj)
4
>>> next(iter_obj)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

PEP 3134 -- Exception Chaining and Embedded Tracebacks （python 3.0）

https://www.python.org/dev/peps/pep-3134/

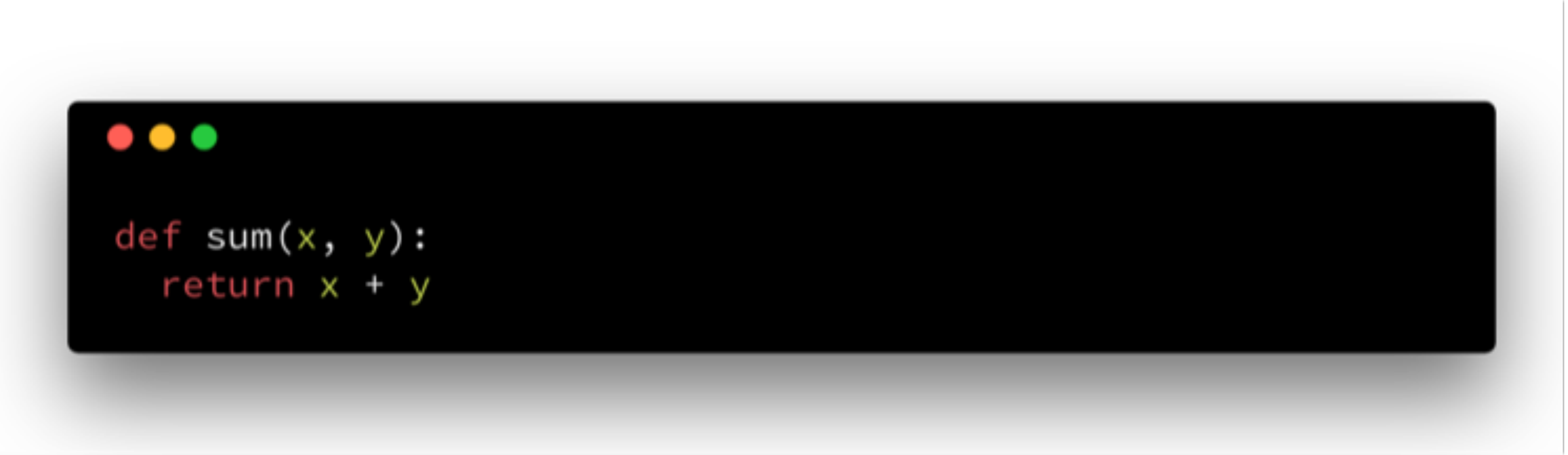
Type Annotations

List of PEPS

- PEP 3107 -- Function Annotations (3.0)
- PEP 482 -- Literature Overview for Type Hints (draft)
- PEP 483 -- The Theory of Type Hints (draft)
- PEP 484 -- Type Hints (3.5)
- PEP 526 -- Syntax for Variable Annotations (3.6)
- PEP 544 -- Protocols: Structural subtyping (static duck typing) (3.7 draft)

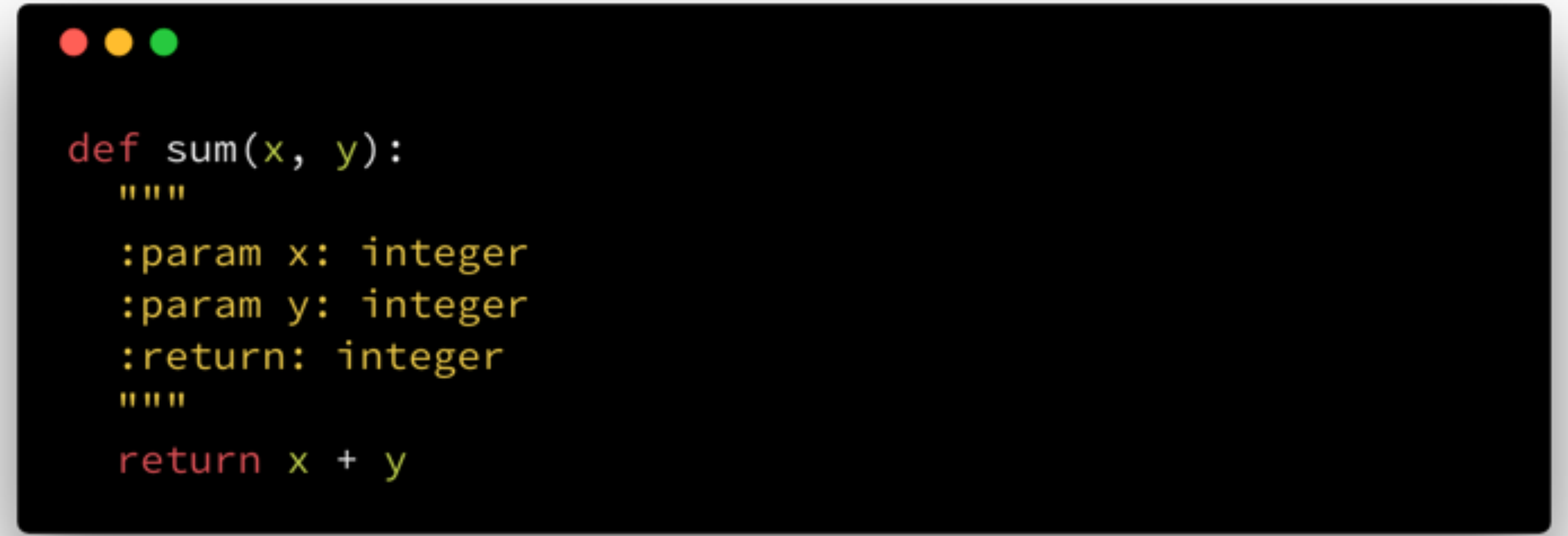
Function Annotations Syntax

Python 2.x



```
def sum(x, y):
  return x + y
```

Python 2.x

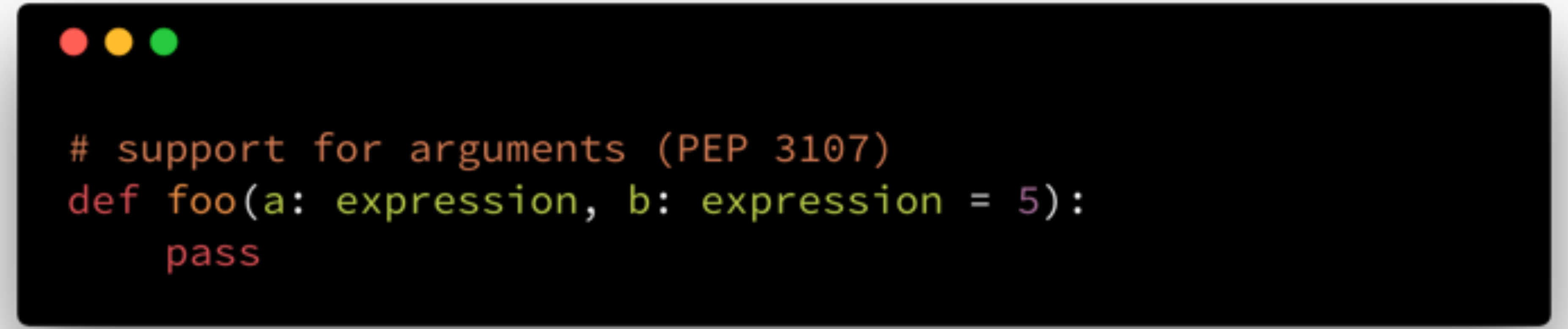


```
def sum(x, y):
  """
  :param x: integer
  :param y: integer
  :return: integer
  """
  return x + y
```

Annotations

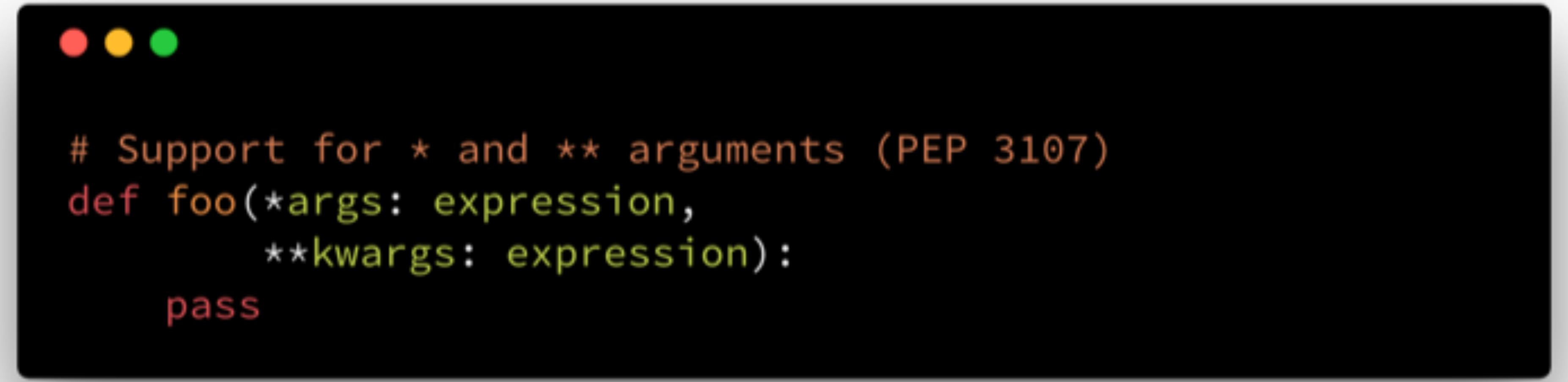
- Aims to provide a single, standard way of specifying function’s parameters and return values. ---- PEP 3107

Syntax



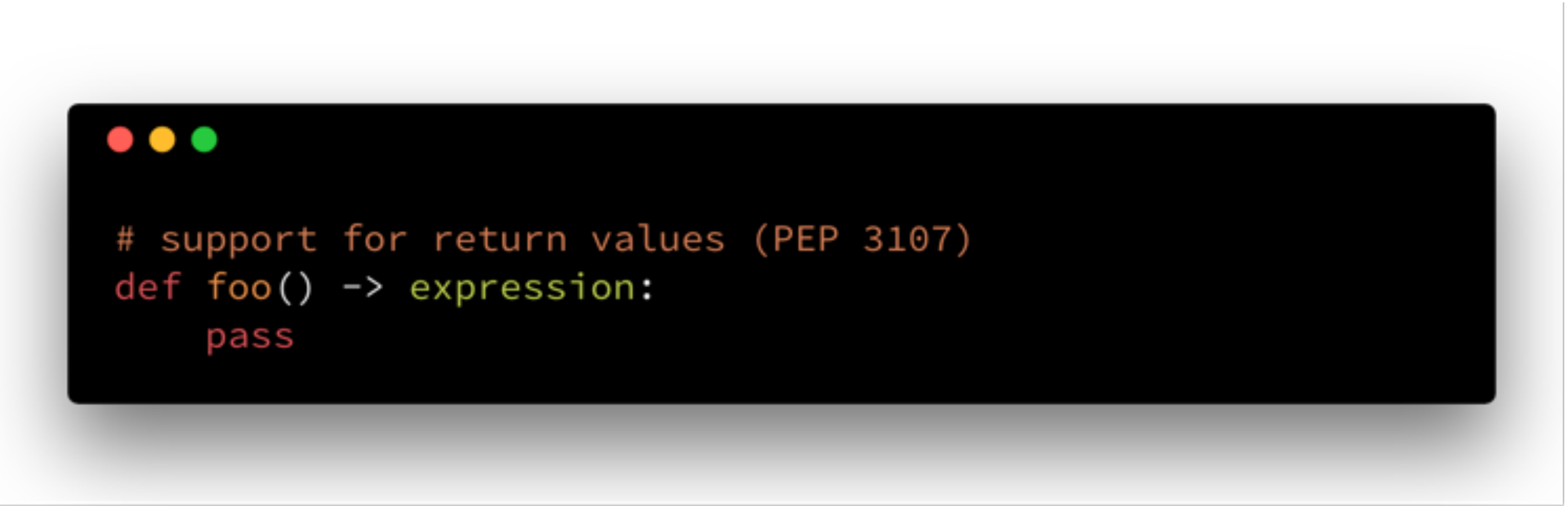
```
# support for arguments (PEP 3107)
def foo(a: expression, b: expression = 5):
    pass
```

Syntax



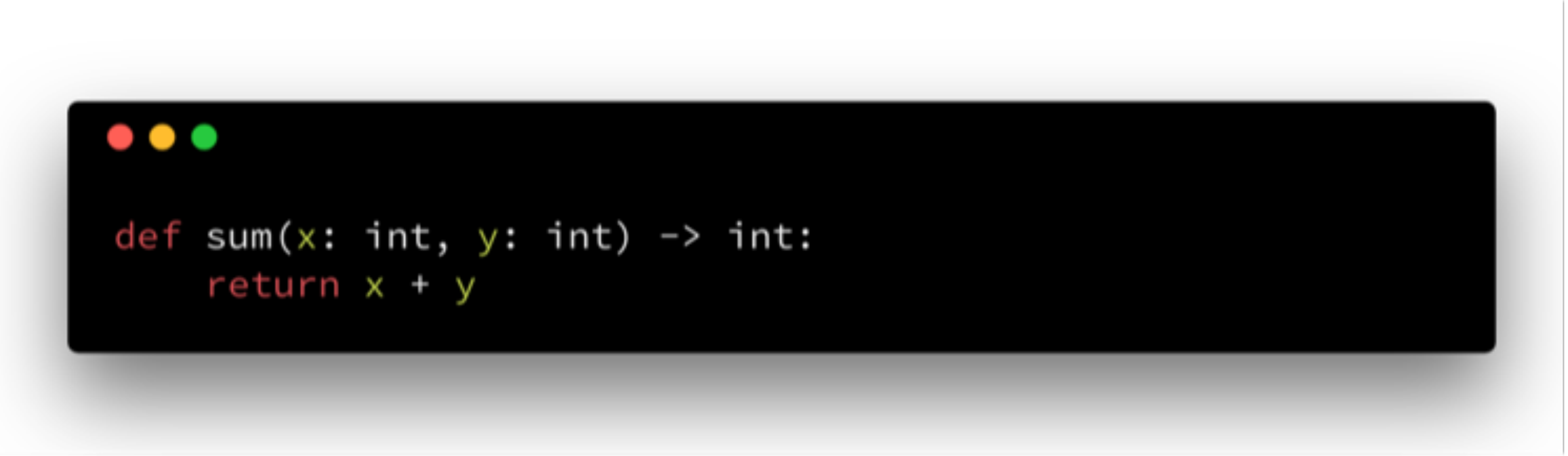
```
# Support for * and ** arguments (PEP 3107)
def foo(*args: expression,
        **kwargs: expression):
    pass
```

Syntax



```
# support for return values (PEP 3107)
def foo() -> expression:
    pass
```

Python 3.x

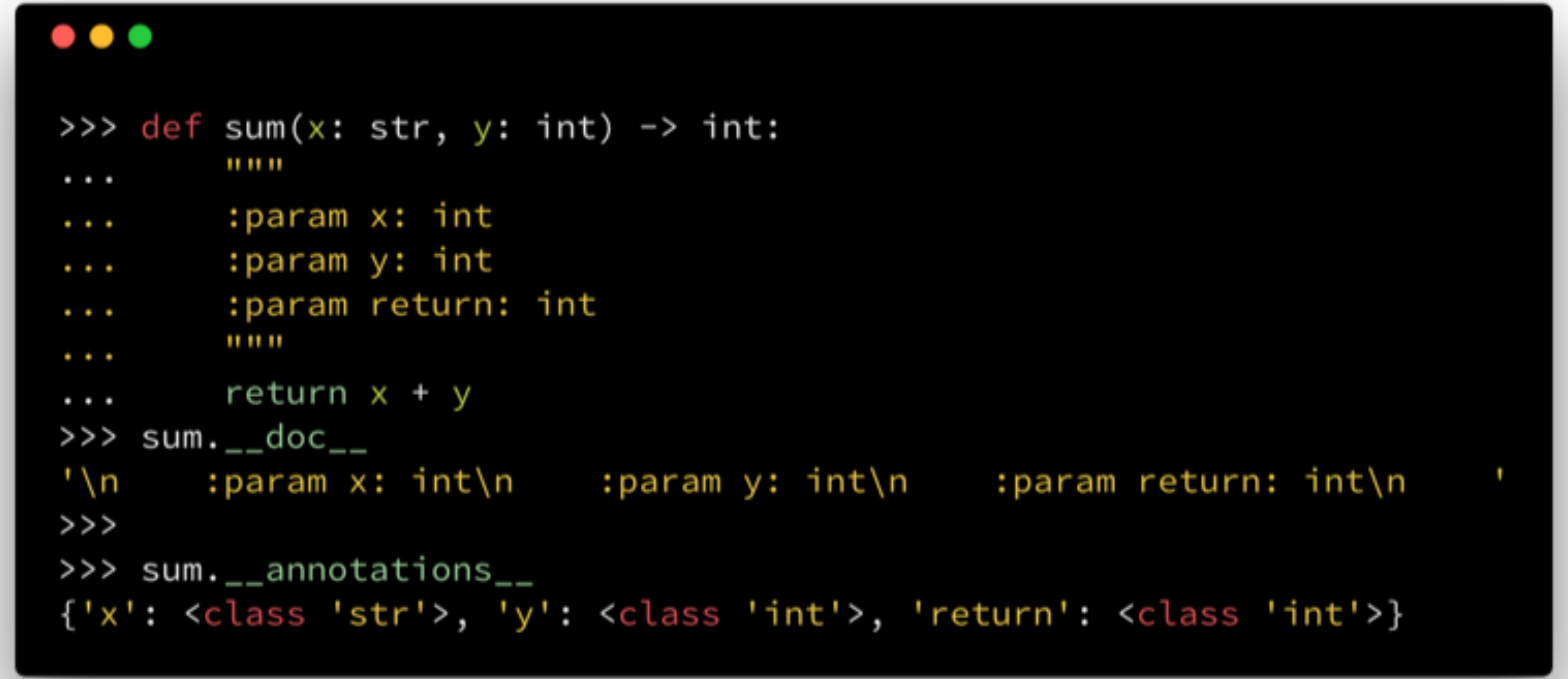


```python
def sum(x: int, y: int) -> int:
    return x + y
```

Python type annotation

- document the types in our code has not effect at runtime(not slower, not faster)
- Mark types for functions does not actually check anything, need 3rd party tools (mypy) for that.

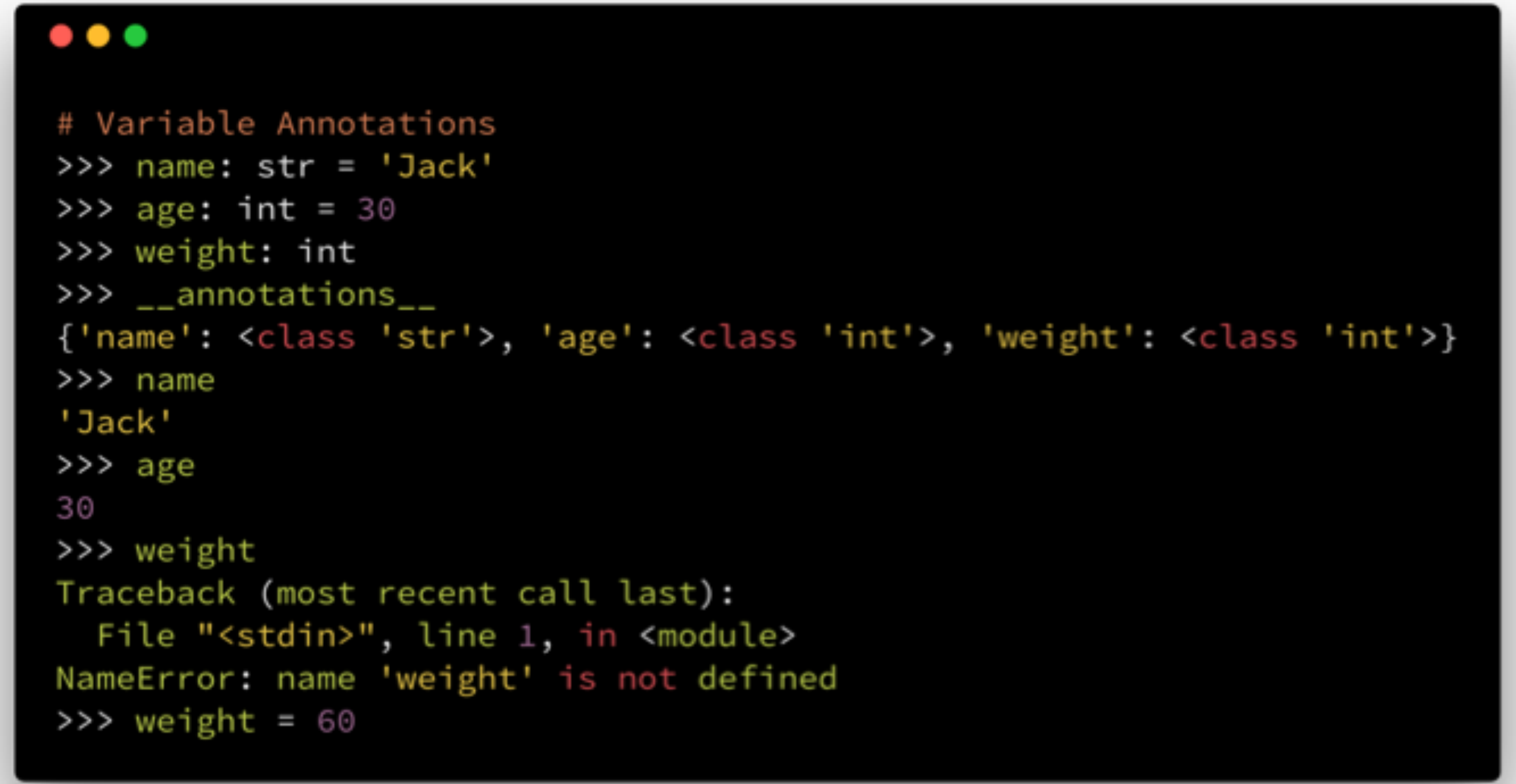
__annotations__



```
>>> def sum(x: str, y: int) -> int:
...     """
...     :param x: int
...     :param y: int
...     :param return: int
...     """
...     return x + y
>>> sum.__doc__
'\n    :param x: int\n    :param y: int\n    :param return: int\n    '
>>>
>>> sum.__annotations__
{'x': <class 'str'>, 'y': <class 'int'>, 'return': <class 'int'>}
```

Variable Annotations

Syntax



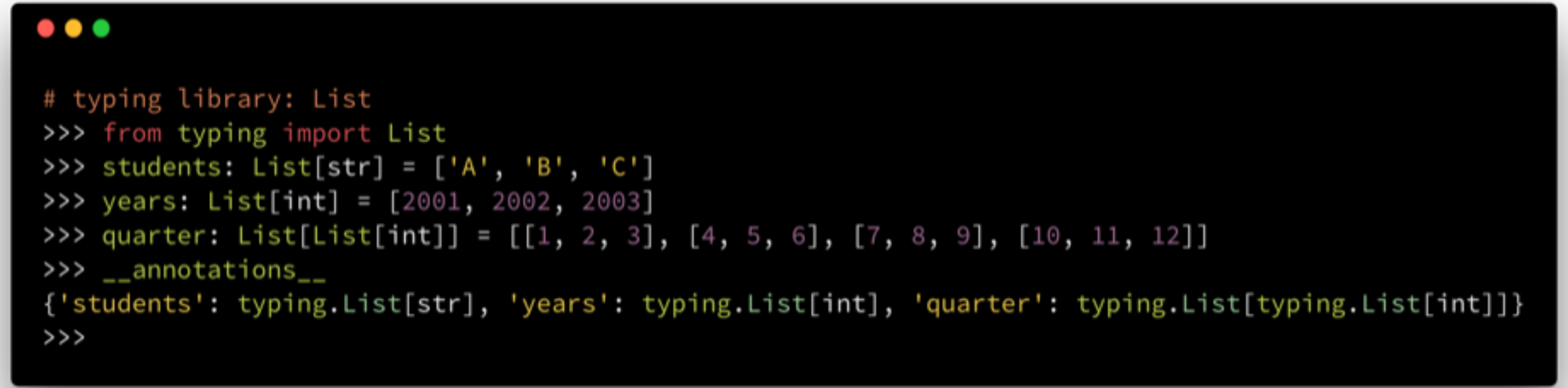
```
# Variable Annotations
>>> name: str = 'Jack'
>>> age: int = 30
>>> weight: int
>>> __annotations__
{'name': <class 'str'>, 'age': <class 'int'>, 'weight': <class 'int'>}
>>> name
'Jack'
>>> age
30
>>> weight
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'weight' is not defined
>>> weight = 60
```

Typing Library

Typing library

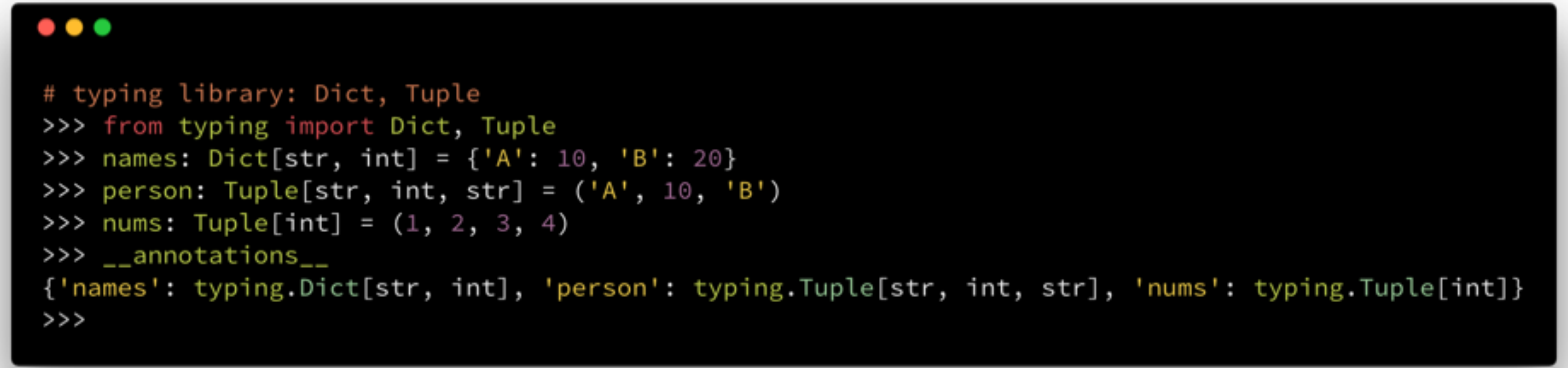
- From Python 3.5, begin to support more complex type for variable annotation like List, Dict, Tuple, Callable, etc.

Typing library - List



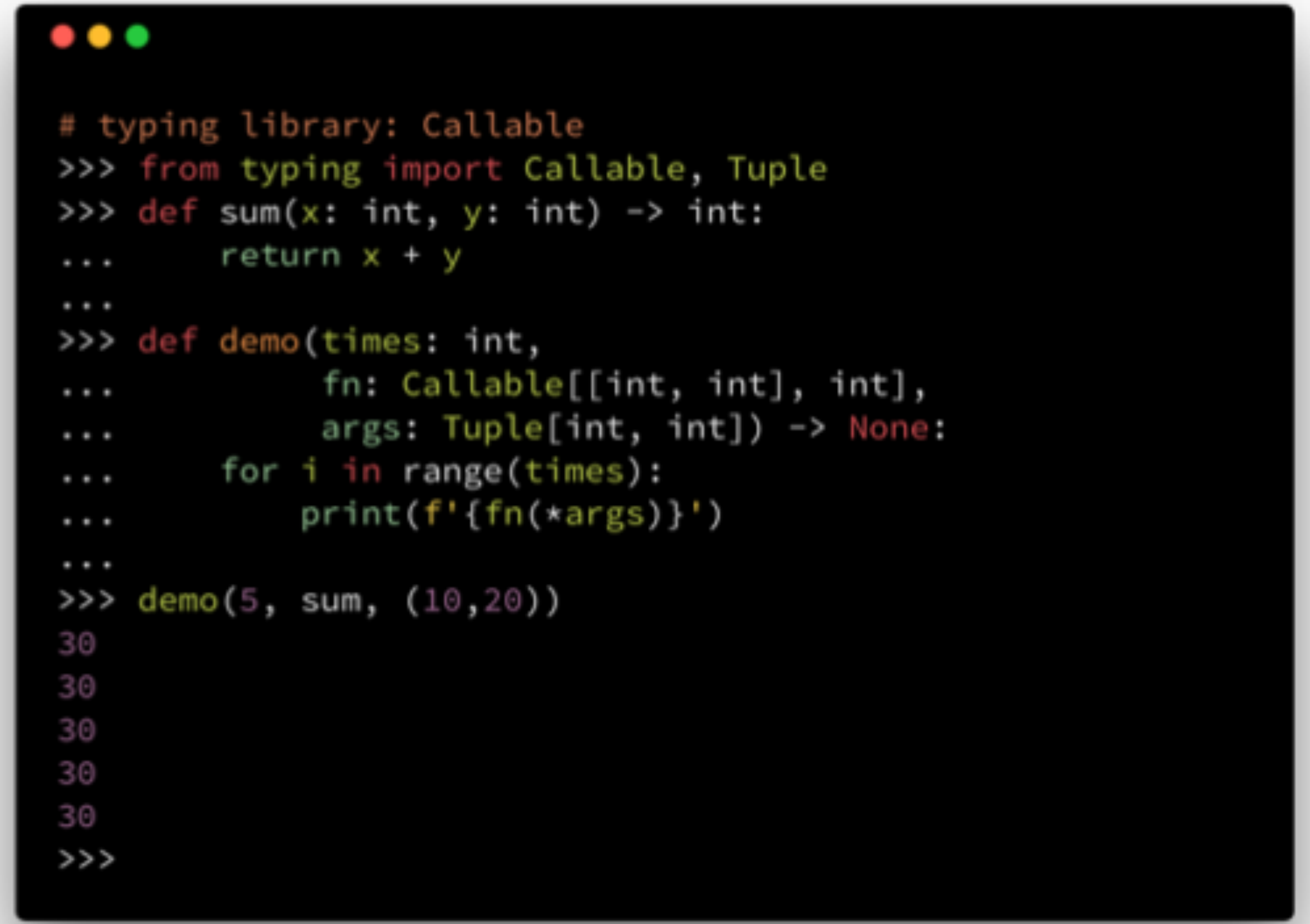
```
# typing library: List
>>> from typing import List
>>> students: List[str] = ['A', 'B', 'C']
>>> years: List[int] = [2001, 2002, 2003]
>>> quarter: List[List[int]] = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
>>> __annotations__
{'students': typing.List[str], 'years': typing.List[int], 'quarter': typing.List[typing.List[int]]}
>>>
```

Typing library – Dict, Tuple



```
# typing library: Dict, Tuple
>>> from typing import Dict, Tuple
>>> names: Dict[str, int] = {'A': 10, 'B': 20}
>>> person: Tuple[str, int, str] = ('A', 10, 'B')
>>> nums: Tuple[int] = (1, 2, 3, 4)
>>> __annotations__
{'names': typing.Dict[str, int], 'person': typing.Tuple[str, int, str], 'nums': typing.Tuple[int]}
>>>
```

Typing library - Callable



```
# typing library: Callable
>>> from typing import Callable, Tuple
>>> def sum(x: int, y: int) -> int:
...     return x + y
...
>>> def demo(times: int,
...          fn: Callable[[int, int], int],
...          args: Tuple[int, int]) -> None:
...     for i in range(times):
...         print(f'{fn(*args)}')
...
>>> demo(5, sum, (10,20))
30
30
30
30
30
>>>
```

Mypy demo

Asyncio

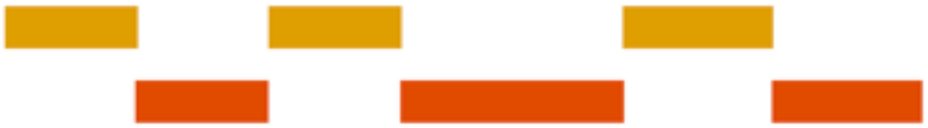
— Asynchronous I/O, event loop, coroutines and tasks

This module provides infrastructure for writing **single-threaded concurrent** code using **coroutines**, multiplexing I/O access over sockets and other resources, running network clients and servers, and other related primitives.

Definitions #1

Concurrency

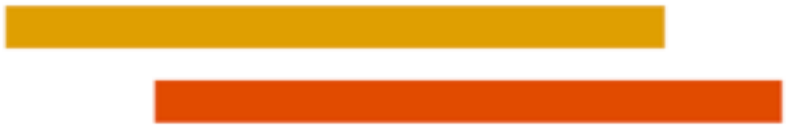
- Tasks start, run, and complete in overlapping time periods.



Coroutines

Parallelism

- Tasks run simultaneously.

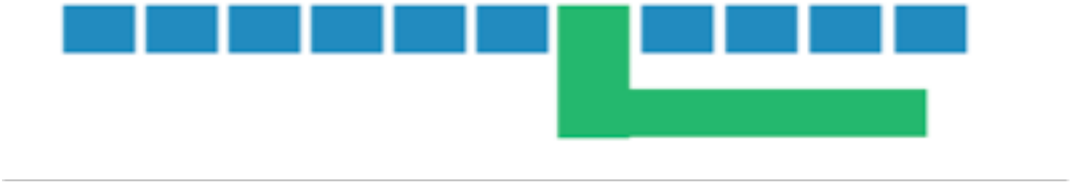


Threads/processes + multicore

Definitions #2

Asynchronous

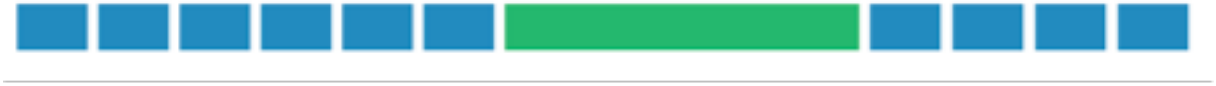
- No need to wait before proceeding.



Overal duration shorter

Synchronous(sequential)

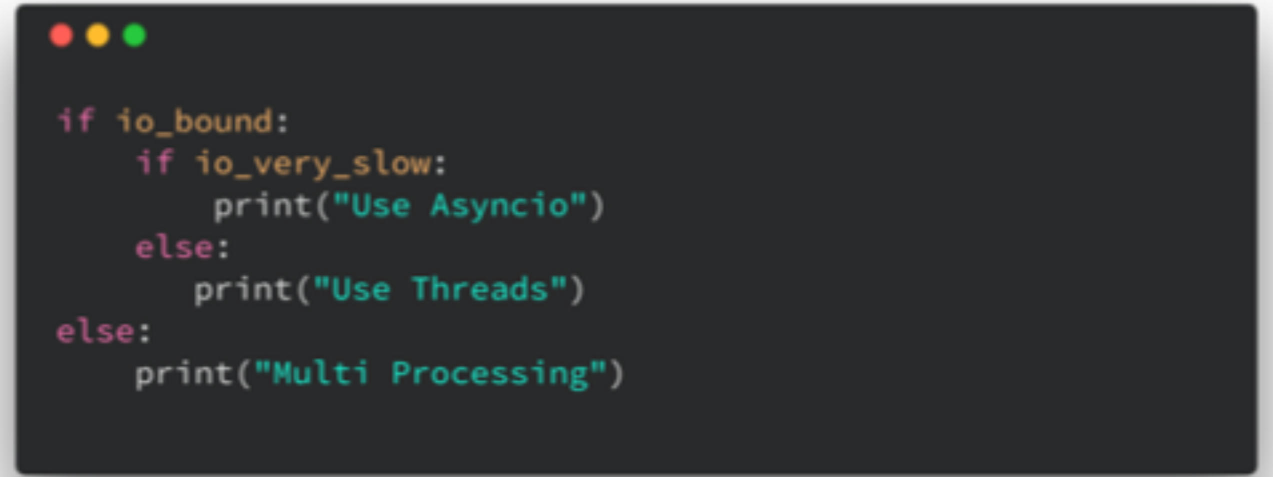
- Must complete before proceeding.



Overal duration longer

Making the right choice

- CPU Bound => Multi Processing
- I/O Bound, Fast I/O, Limited Number of Connections => Multi Threading
- I/O Bound, Slow I/O, Many connections => Asyncio



```
if io_bound:
    if io_very_slow:
        print("Use Asyncio")
    else:
       print("Use Threads")
else:
    print("Multi Processing")
```

Reference for asyncio

- http://masnun.rocks/2016/10/06/async-python-the-different-forms-of-concurrency/ Async Python: The Different Forms of Concurrency python
- https://www.youtube.com/watch?v=c5wodlqGK-M Coroutine Concurrency in Python 3 with asyncio - Robert Smallshire
- https://docs.python.org/3.6/library/asyncio.html
- https://magic.io/blog/uvloop-blazing-fast-python-networking/ uvloop: Blazing fast Python networking
- http://sanic.readthedocs.io/en/latest/