Exception Handling, Class, Iterator & Generator

SI100B Fall 2020 Tutorial Week 3

Wu Daqian

Simple Review

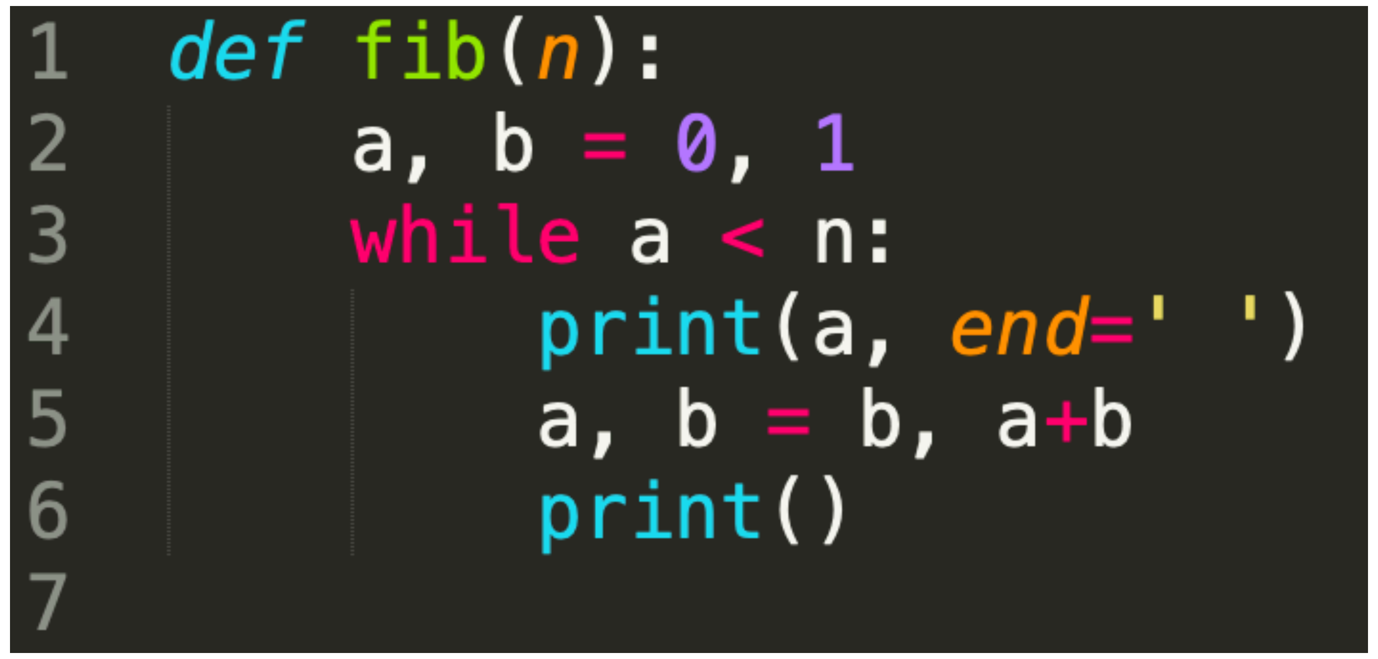
- Loop
- Function and scope
- Insight into List
- Lambda Expression

Loop

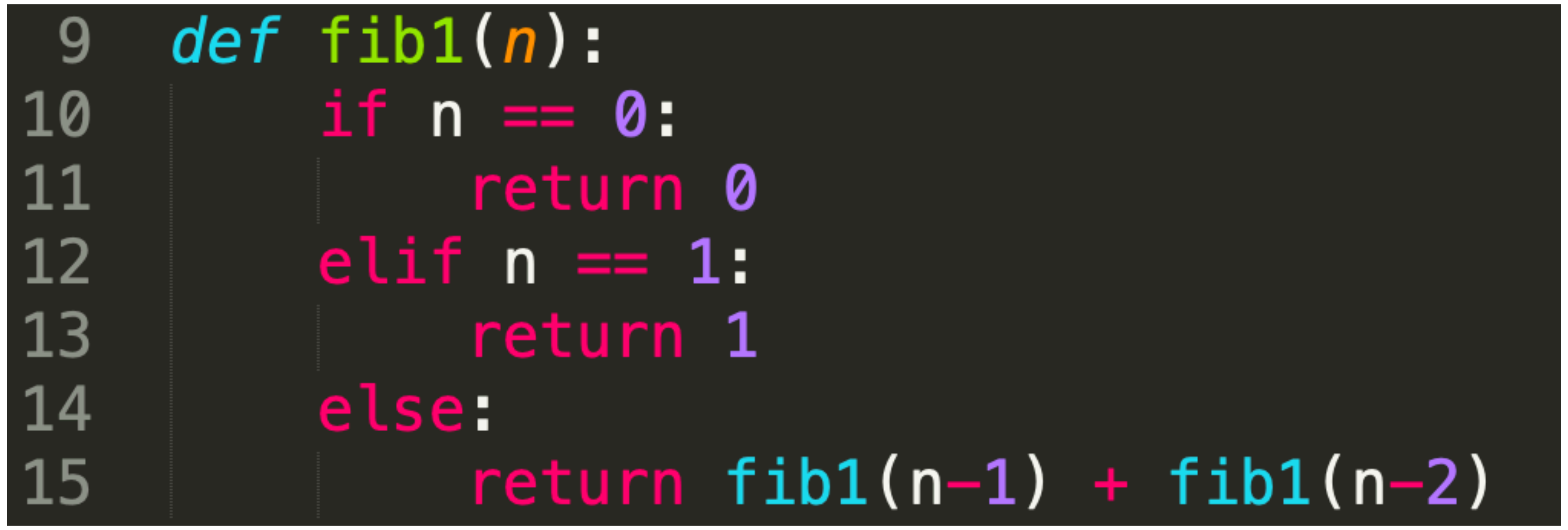
“Use your loop in a smart way!”

Function & Scope

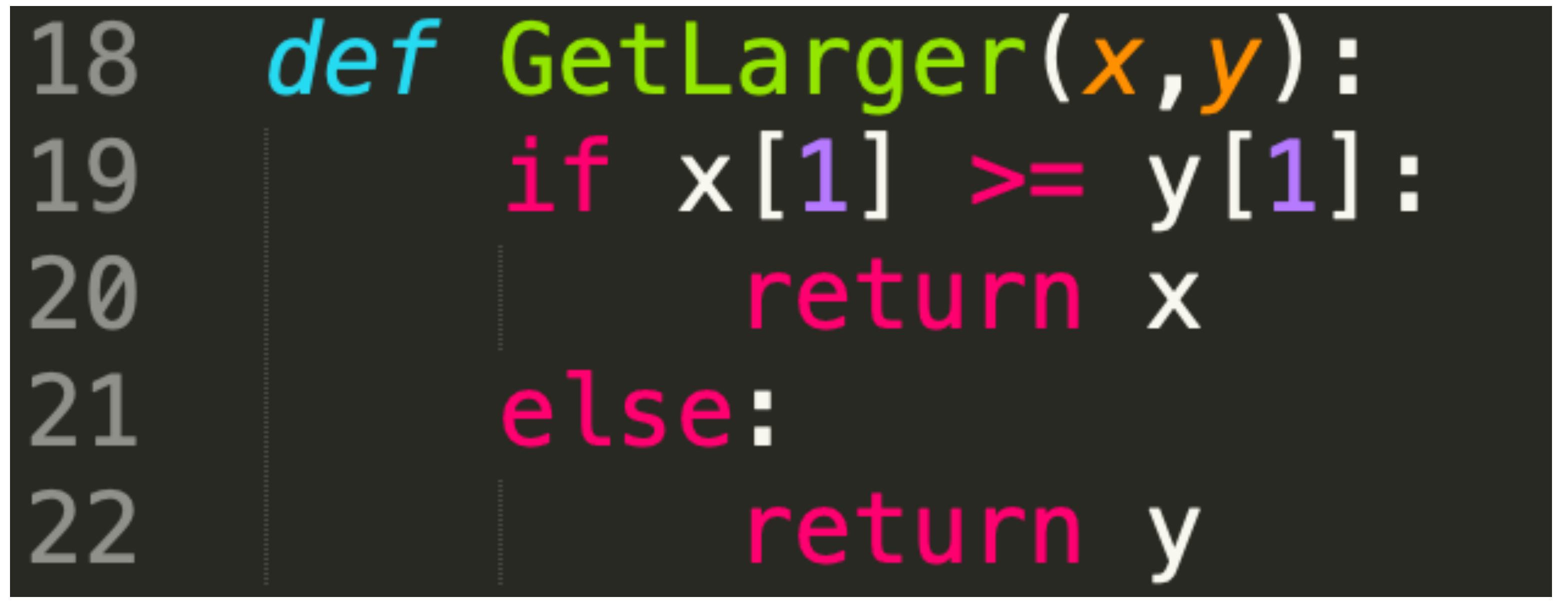
- Function
  - Definition
  - Invocation
- Scope
  - Local
  - Non-local
  - Global
  - Name resolution



```python
def fib(n):
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
        print()

```



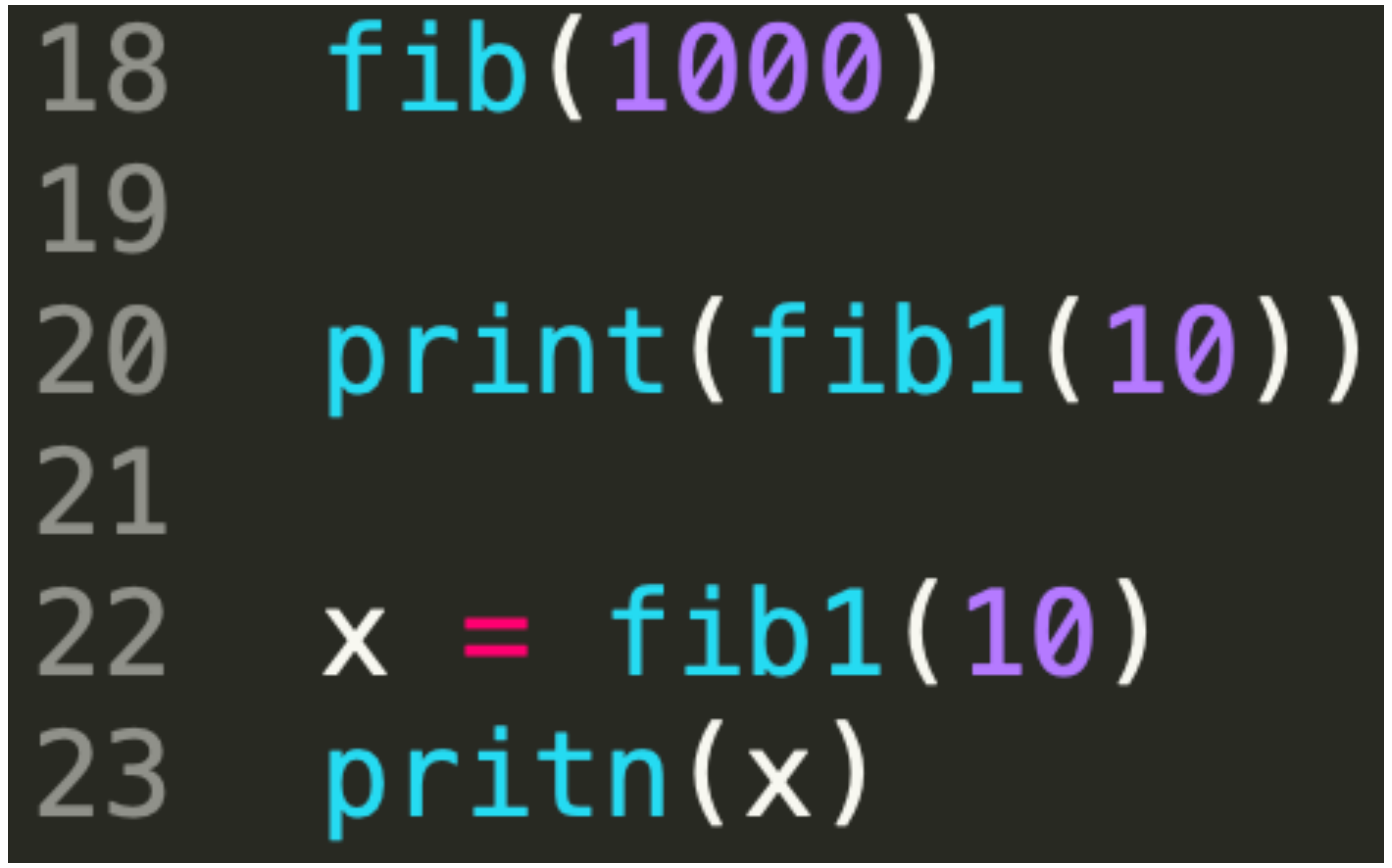
```python
def fib1(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib1(n-1) + fib1(n-2)
```



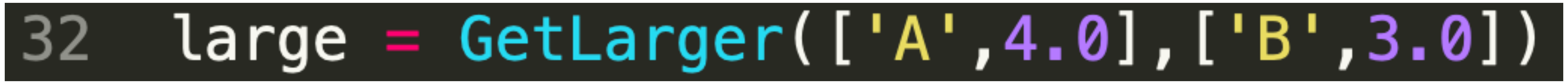
```python
def GetLarger(x,y):
    if x[1] >= y[1]:
        return x
    else:
        return y
```

Function & Scope

- Function
  - Definition
  - Invocation
- Scope
  - Local
  - Non-local
  - Global
  - Name resolution



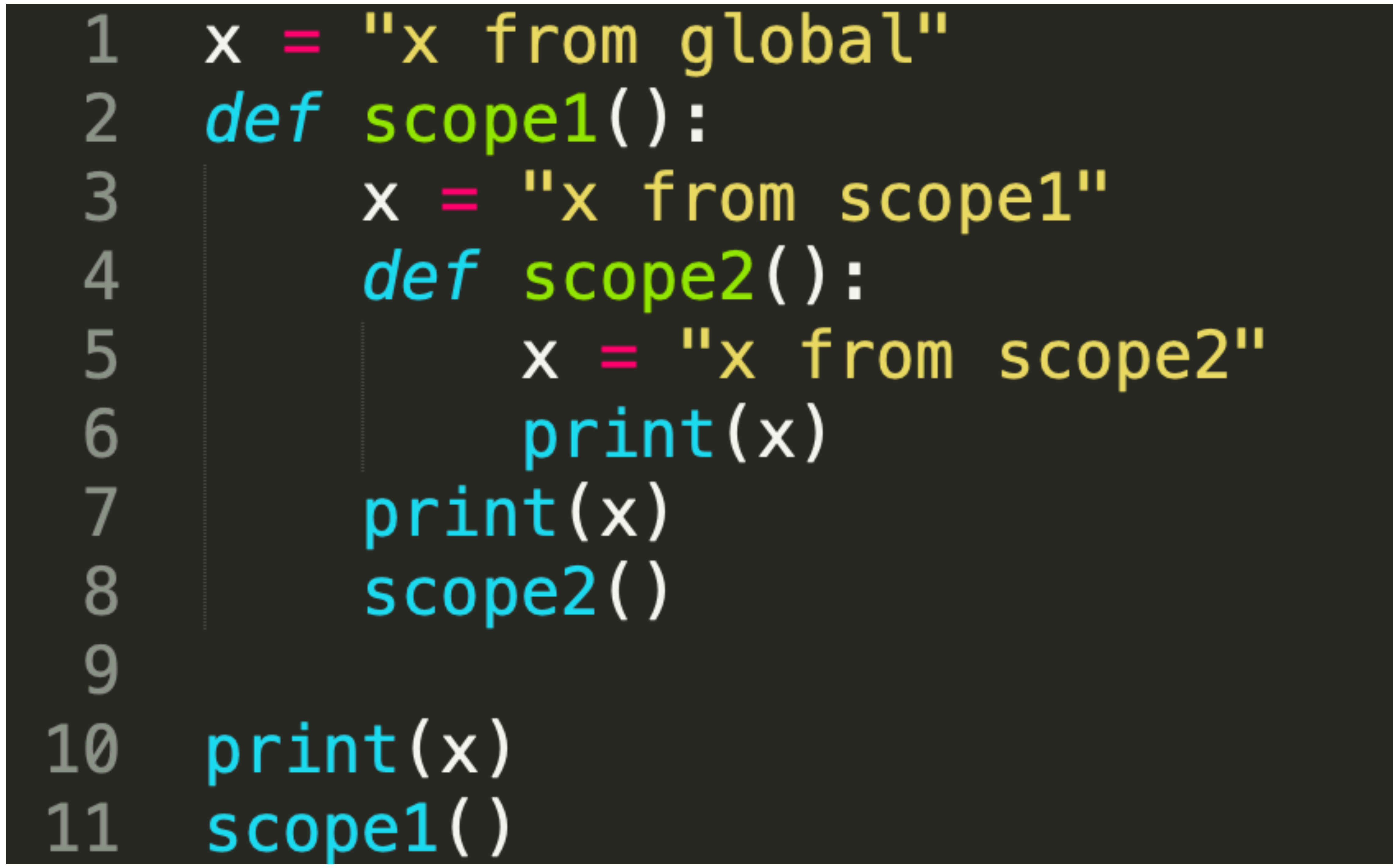
```
fib(1000)

print(fib1(10))

x = fib1(10)
pritn(x)
```



```
large = GetLarger(['A',4.0],['B',3.0])
```

Function & Scope

- Function
  - Definition
  - Invocation
- Scope
  - Local
  - Non-local
  - Global
  - Name resolution



```python
x = "x from global"
def scope1():
    x = "x from scope1"
    def scope2():
        x = "x from scope2"
        print(x)
    print(x)
    scope2()

print(x)
scope1()
```

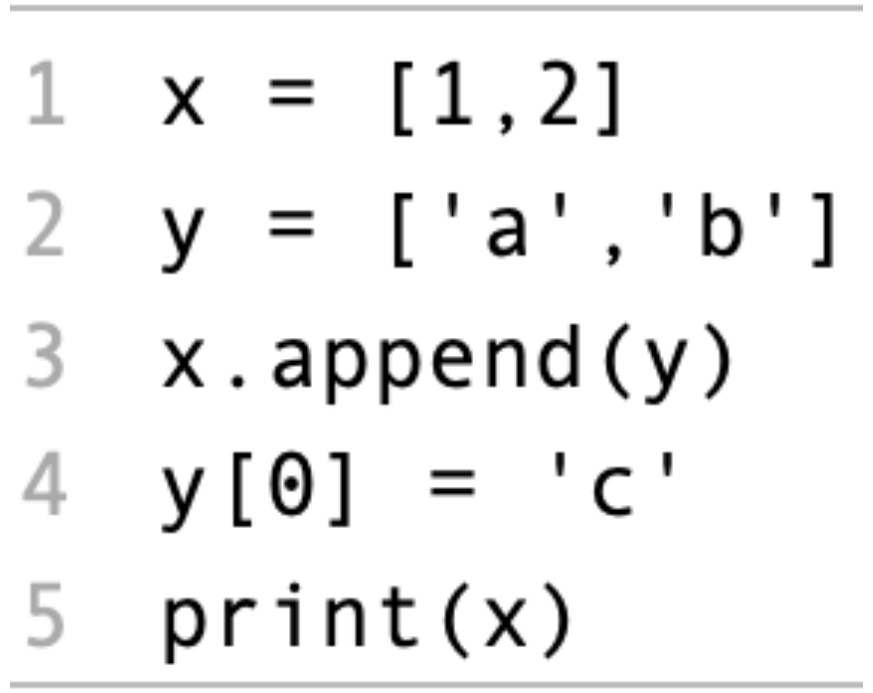
Function & Scope

- Function
  - Definition
  - Invocation
- Scope
  - Local
  - Non-local
  - Global
  - Name resolution

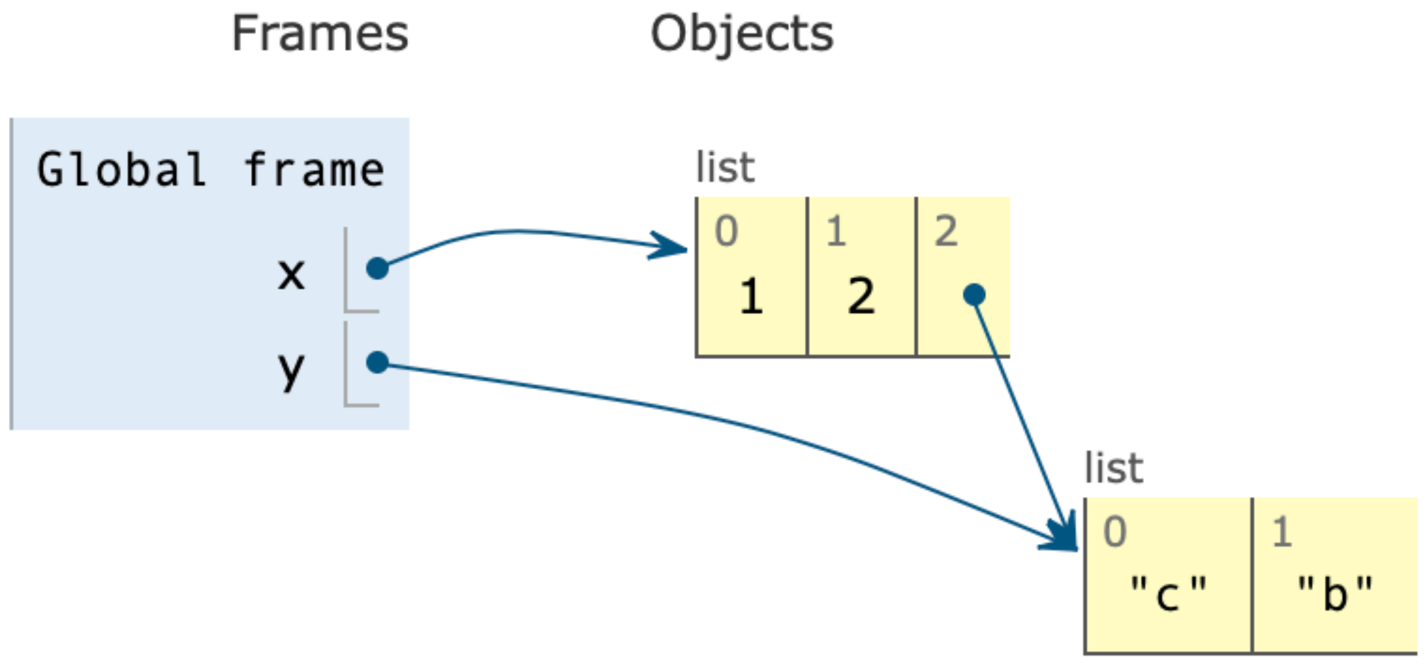
- **Local (or function) scope** is the code block or body of any Python function or `lambda` expression. This Python scope contains the names that you define inside the function. These names will only be visible from the code of the function. It's created at function call, *not* at function definition, so you'll have as many different local scopes as function calls. This is true even if you call the same function multiple times, or recursively. Each call will result in a new local scope being created.
- **Enclosing (or nonlocal) scope** is a special scope that only exists for nested functions. If the local scope is an inner or nested function, then the enclosing scope is the scope of the outer or enclosing function. This scope contains the names that you define in the enclosing function. The names in the enclosing scope are visible from the code of the inner and enclosing functions.
- **Global (or module) scope** is the top-most scope in a Python program, script, or module. This Python scope contains all of the names that you define at the top level of a program or a module. Names in this Python scope are visible from everywhere in your code.
- **Built-in scope** is a special Python scope that's created or loaded whenever you run a script or open an interactive session. This scope contains names such as keywords, functions, exceptions, and other attributes that are built into Python. Names in this Python scope are also available from everywhere in your code. It's automatically loaded by Python when you run a program or script.

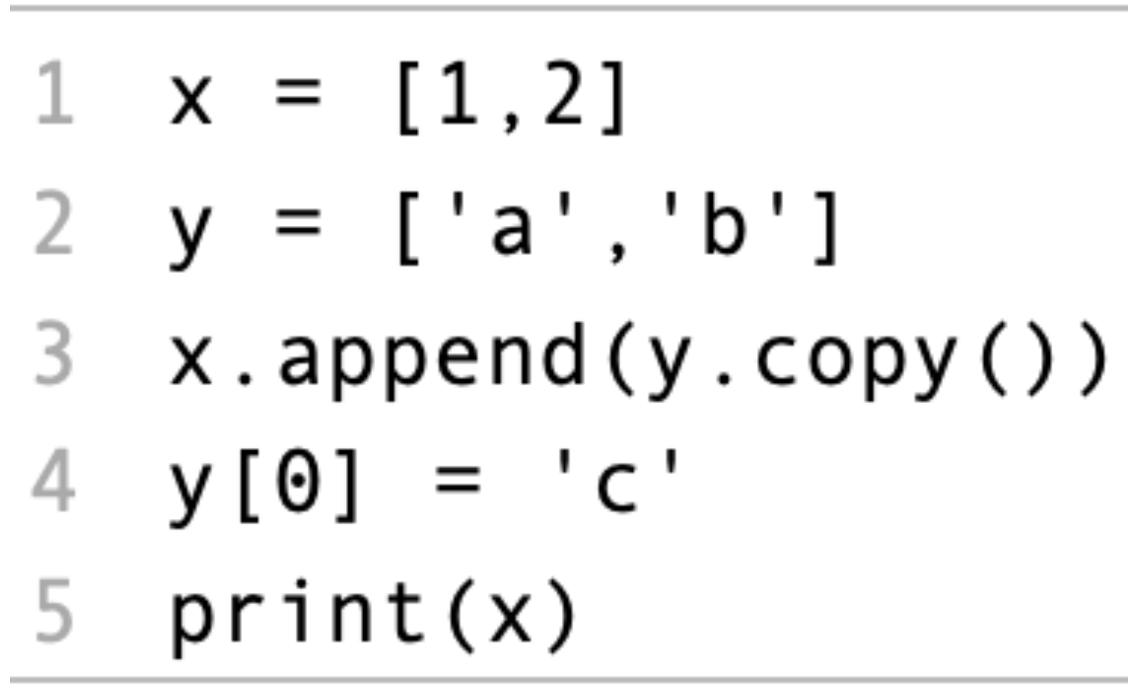
Insight into List

- List
  - __add__
  - extend
  - append
  - __mul__

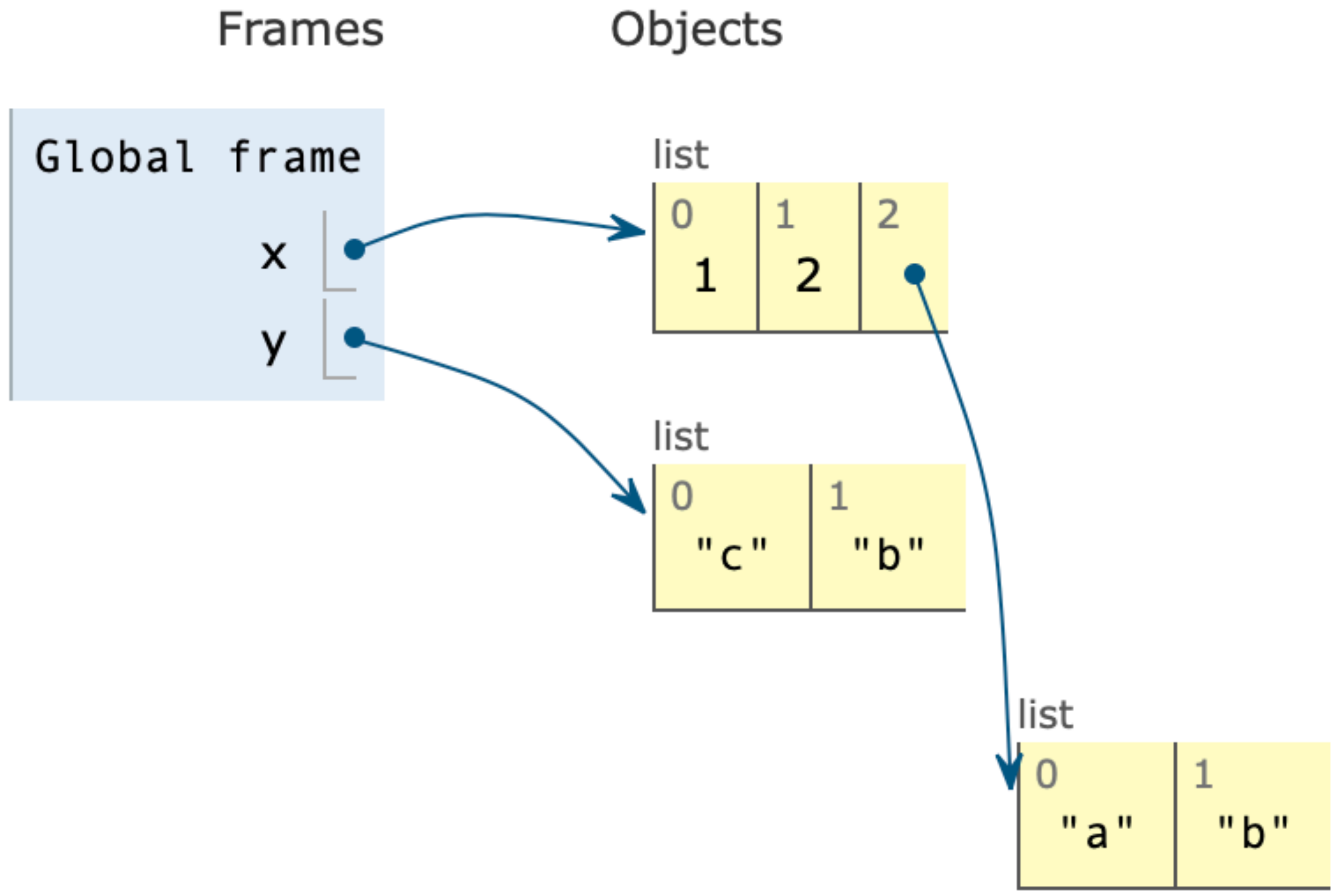


```
x = [1,2]
y = ['a','b']
x.append(y)
y[0] = 'c'
print(x)
```





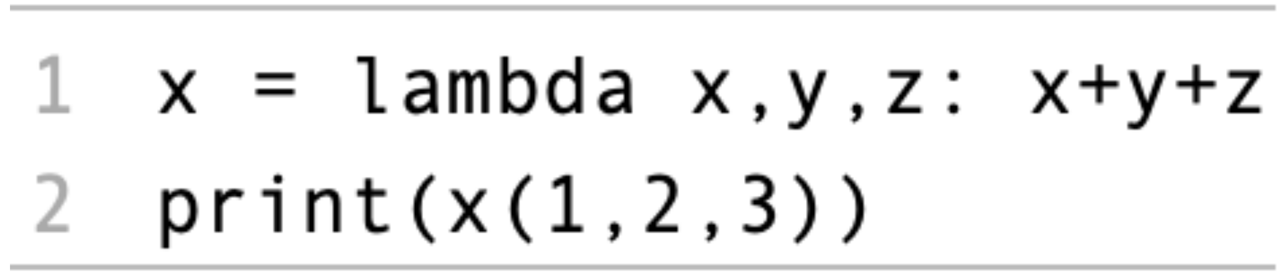
```
x = [1,2]
y = ['a','b']
x.append(y.copy())
y[0] = 'c'
print(x)
```



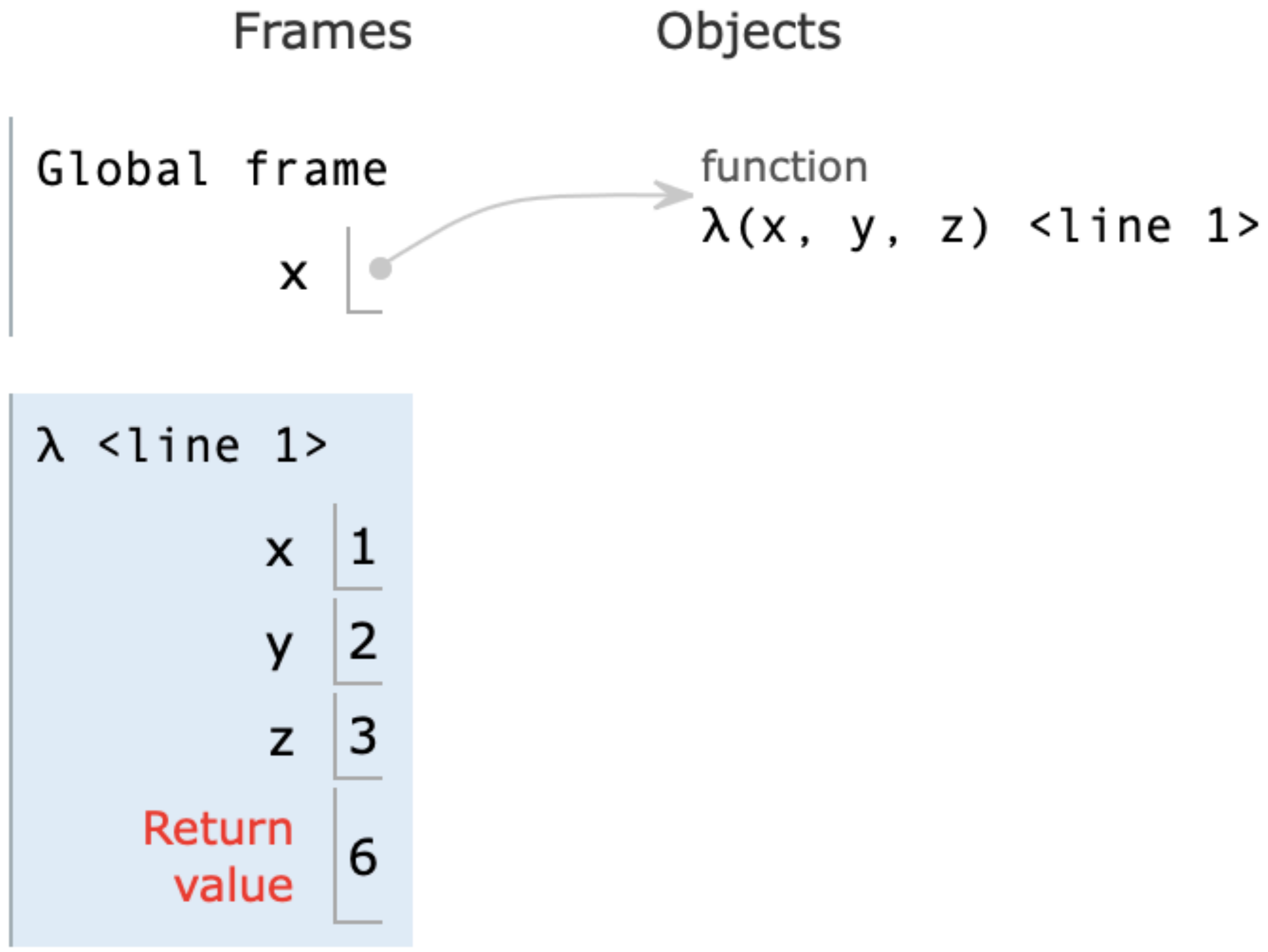
Lambda Expression

Lambda expression can be used to defined small anonymous functions

- **Parameter_list: is a list of parameters $p_1,p_2,\ldots,p_n$**
- **The anonymous function return the value of expr**



```
x = lambda x,y,z: x+y+z
print(x(1,2,3))
```



Exception Handling

Errors and Exceptions

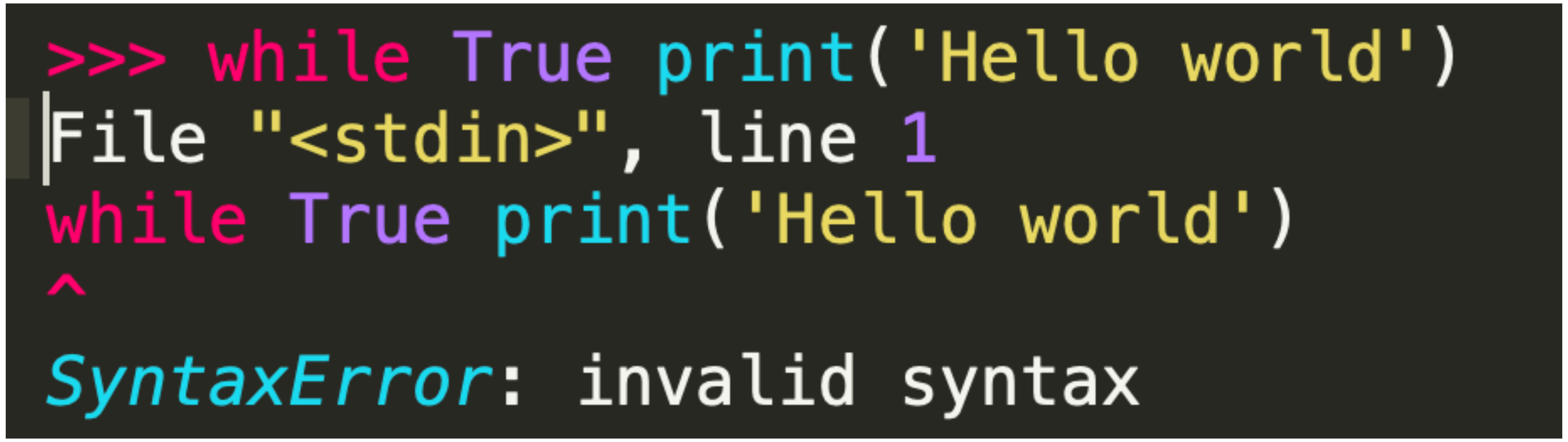
- Error

  SyntaxError,

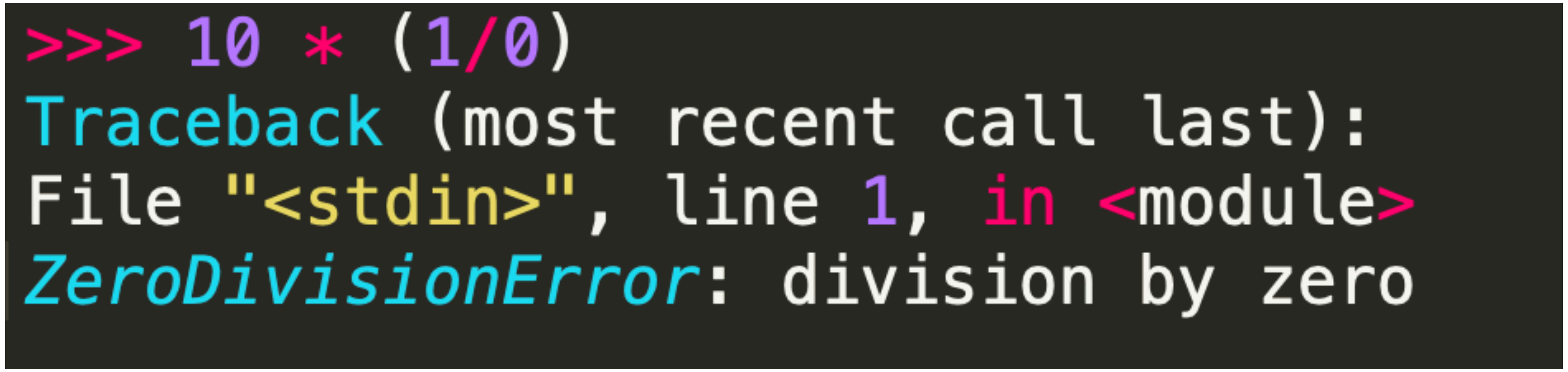
  TypeError,

  …

- Exception



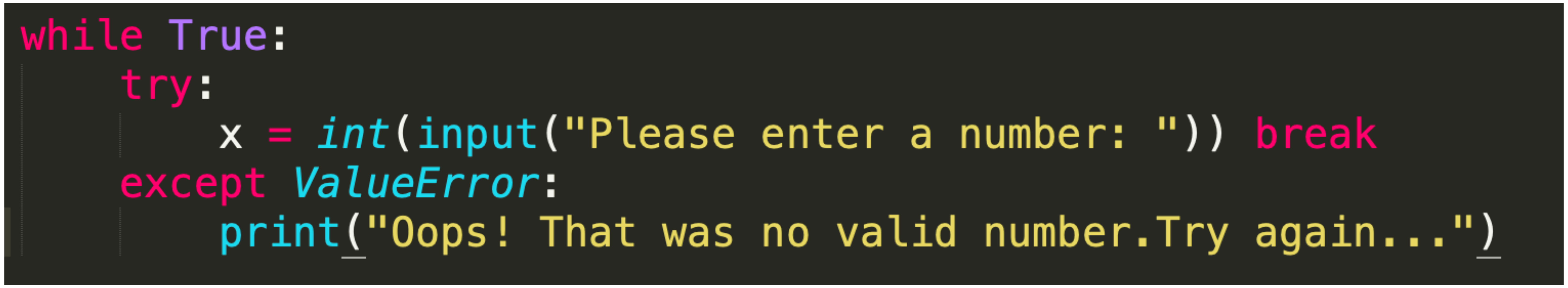
```
>>> while True print('Hello world')
File "<stdin>", line 1
while True print('Hello world')
^
SyntaxError: invalid syntax
```



```
>>> 10 * (1/0)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Handling Exceptions

- Improve robustness and fault tolerance
- User-friendly error message
- Try statement

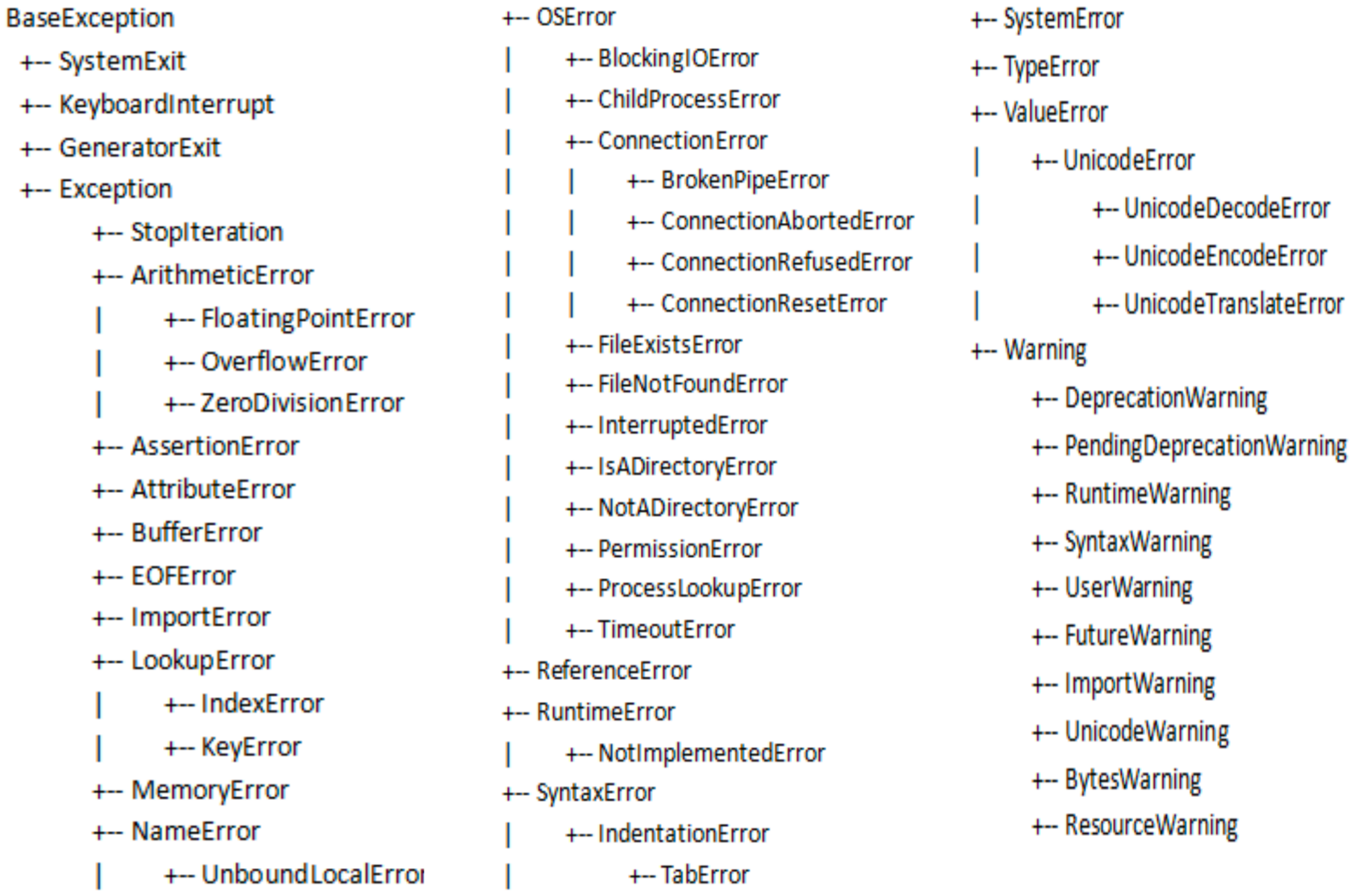


```
while True:
    try:
        x = int(input("Please enter a number: ")) break
    except ValueError:
        print("Oops! That was no valid number.Try again...")
```

If-Else vs. Exception Handling

- It is better to use exception handling than if-else
- Proper use of exception handling, instead of abuse of exception handling
- Catch precise exception
- Proper exception handling for different exception

Exception Hierarchy

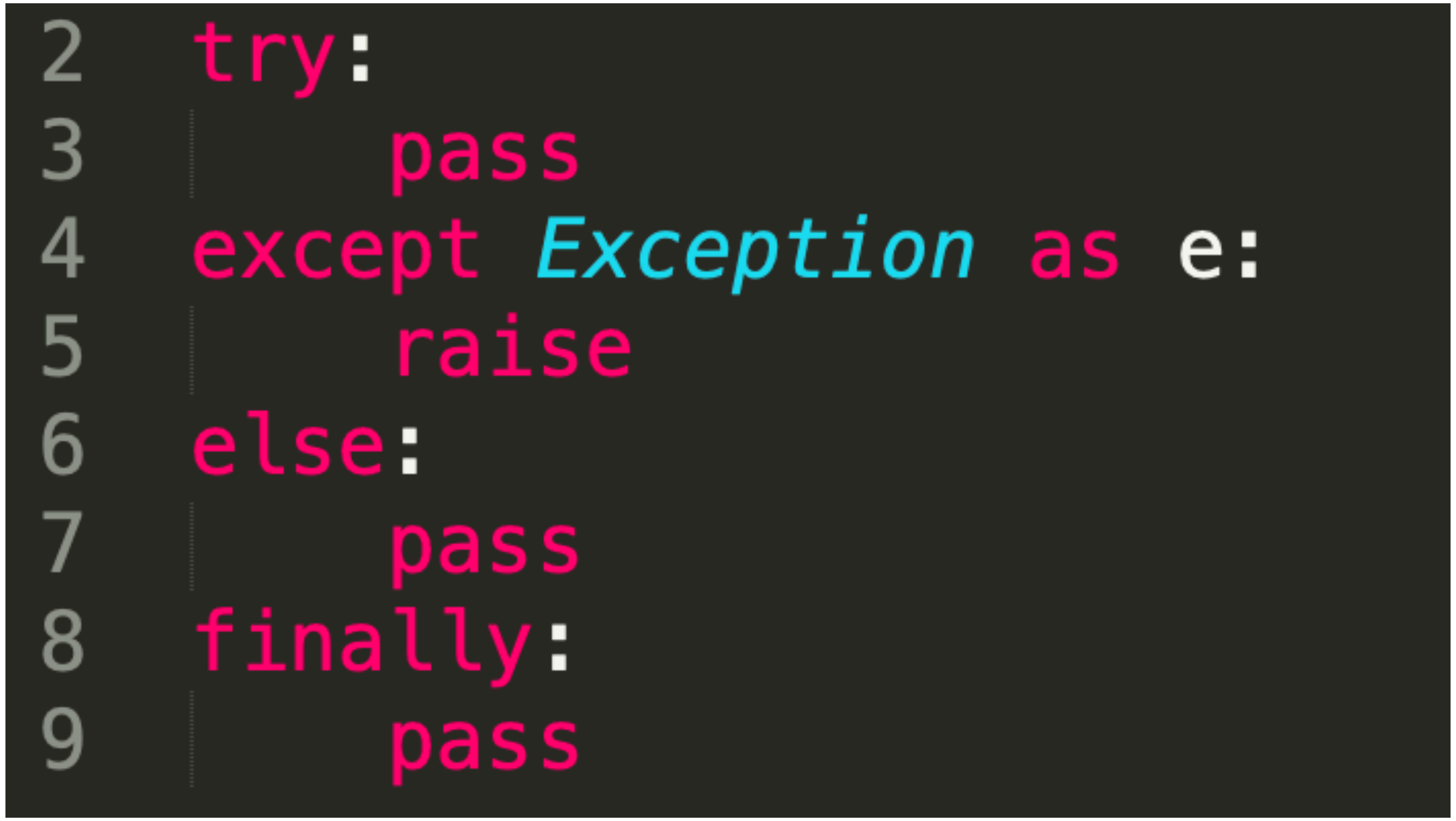


```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

The try statement

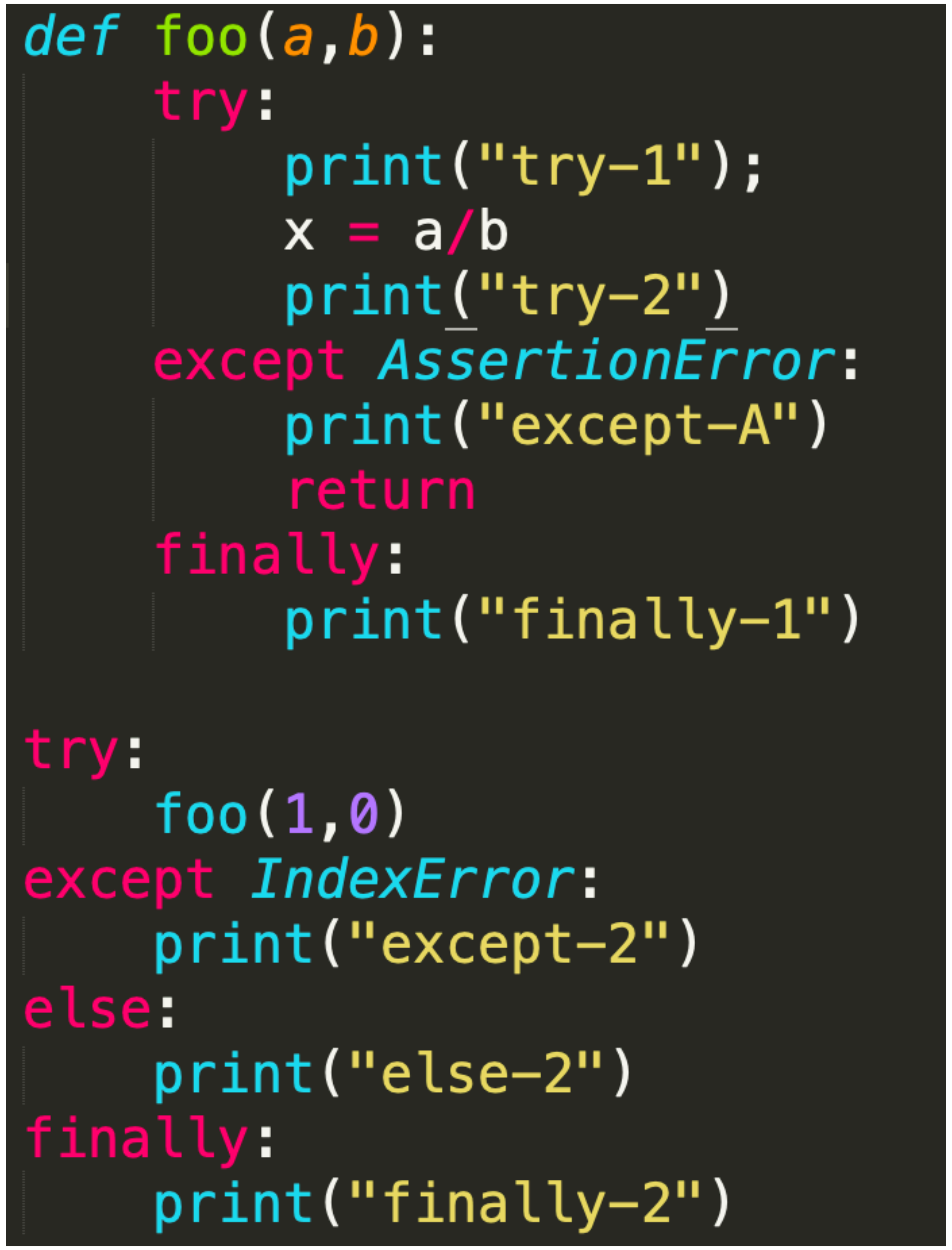
If no exception occurs in the try clause, no exception handler is executed;

But else clause is executed if no return, continue, or break statement was executed in try clause;
Exceptions in the else clause are not handled by the preceding except clauses;

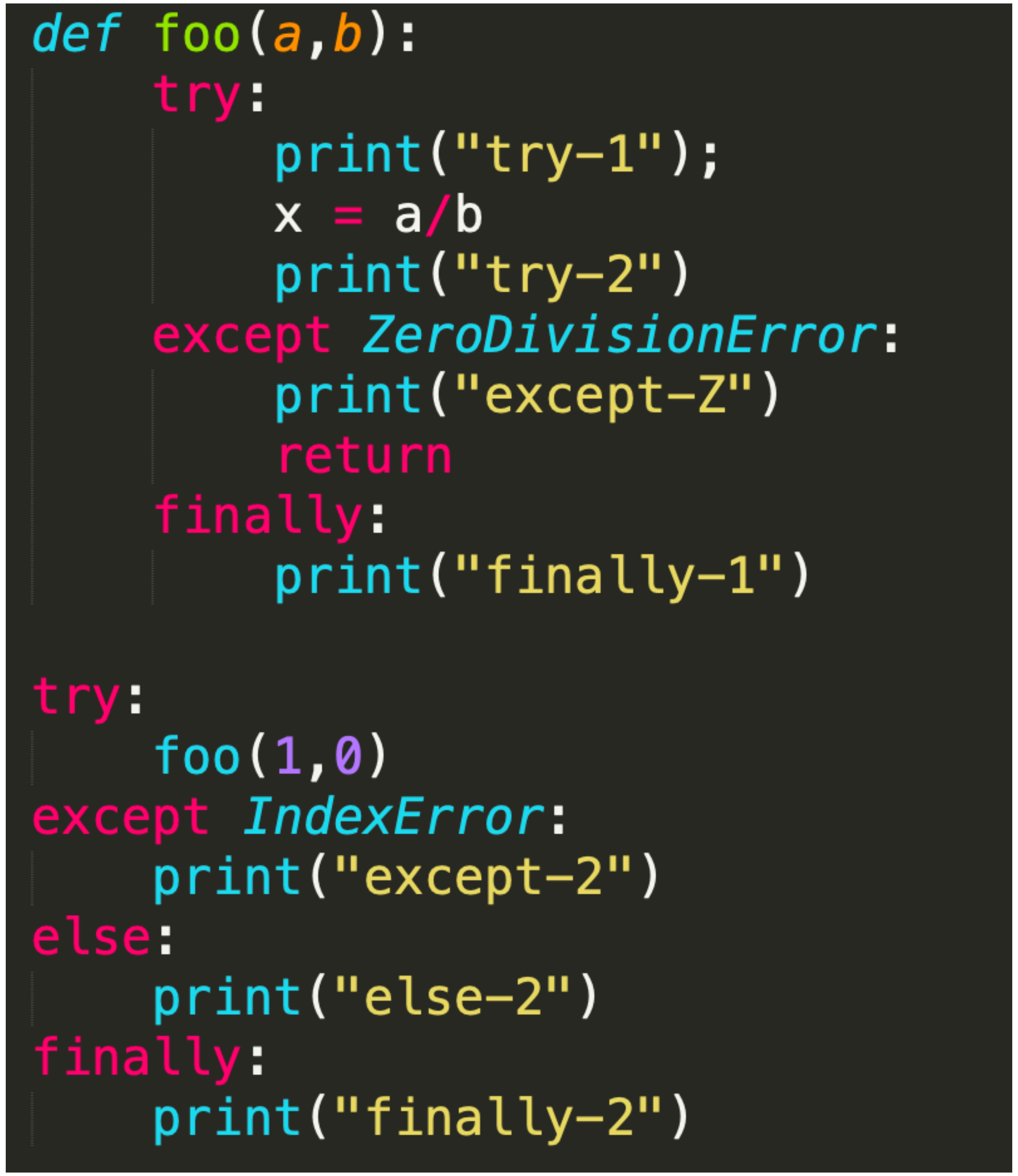


```
try:
    pass
except Exception as e:
    raise
else:
    pass
finally:
    pass
```

The try statement



```python
def foo(a,b):
    try:
        print("try-1");
        x = a/b
        print("try-2")
    except AssertionError:
        print("except-A")
        return
    finally:
        print("finally-1")

try:
    foo(1,0)
except IndexError:
    print("except-2")
else:
    print("else-2")
finally:
    print("finally-2")
```

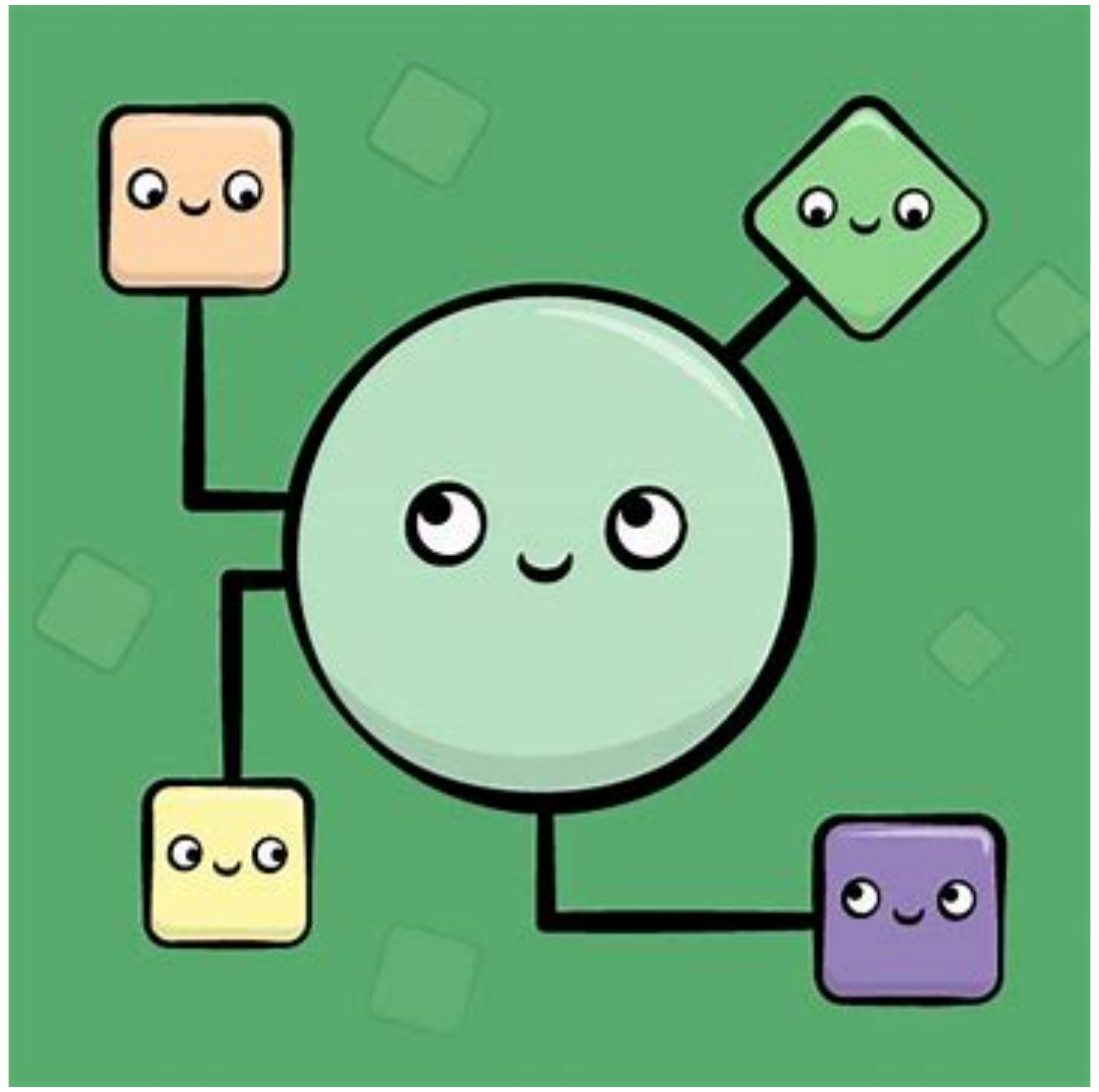


```python
def foo(a,b):
    try:
        print("try-1");
        x = a/b
        print("try-2")
    except ZeroDivisionError:
        print("except-Z")
        return
    finally:
        print("finally-1")

try:
    foo(1,0)
except IndexError:
    print("except-2")
else:
    print("else-2")
finally:
    print("finally-2")
```

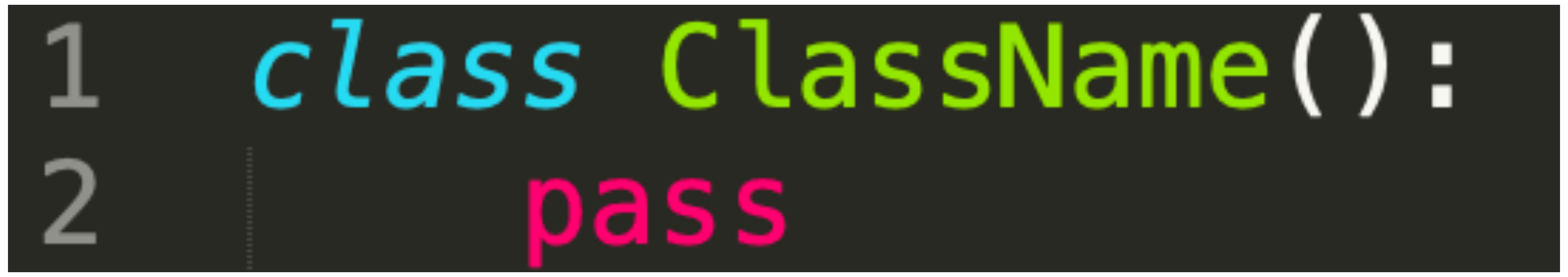
Object Oriented Programming

Object Oriented Programming

- In OOP, code and data are combined into a single entity called a class
  - each instance of a given class is an object of that class type
- Principles of Object-Oriented Programming
  - encapsulation
  - inheritance
  - polymorphism
- Python is Pure OO
  - Everything in Python is an object (excluding keywords)



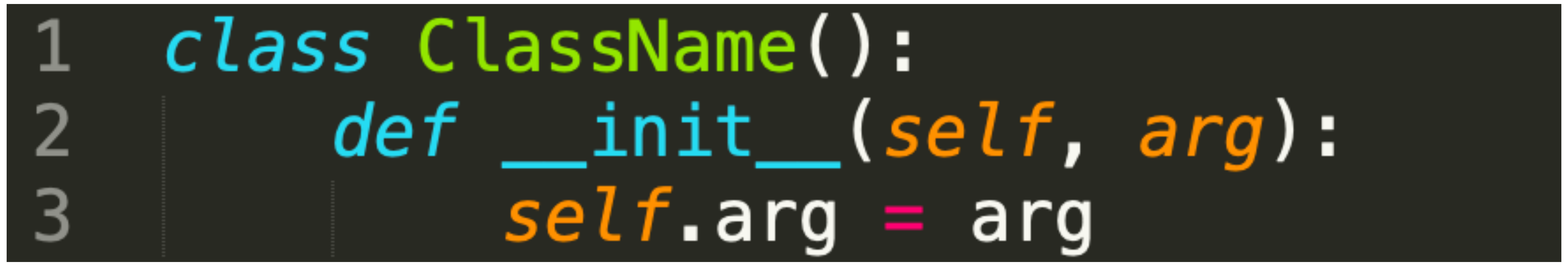
Class



```python
class ClassName():
    pass
```

- A class definition starts with the keyword class
- Following a classname, the first character of the name is usually UPPERCASE
- Then, the colon :
- The class body consists of a sequence of statements and/or function definitions, organized via INDENT

Constructor



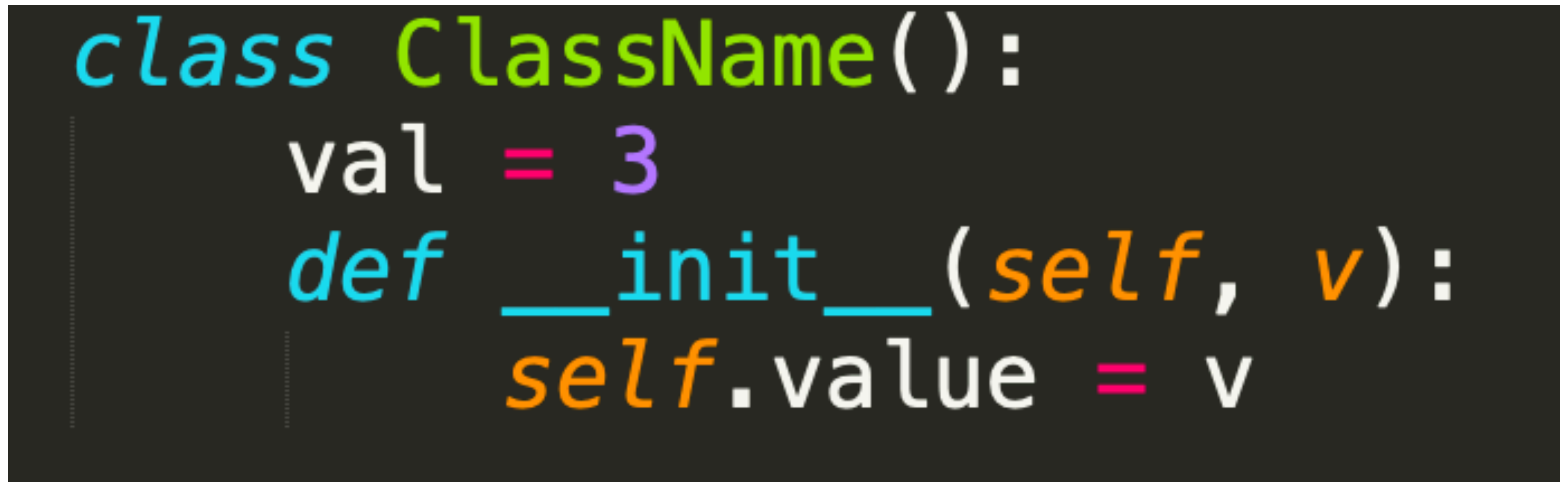
```python
class ClassName():
    def __init__(self, arg):
        self.arg = arg
```

- All the classes have an implicit instance method __init__ as constructor (inherited from the class object)
- It is called after the instance has been created, but before it is returned to the caller
- The arguments are those passed to the class constructor expression
- The first parameter of __init__ is the instance object
- One can override __init__ in user-defined classes for initialization

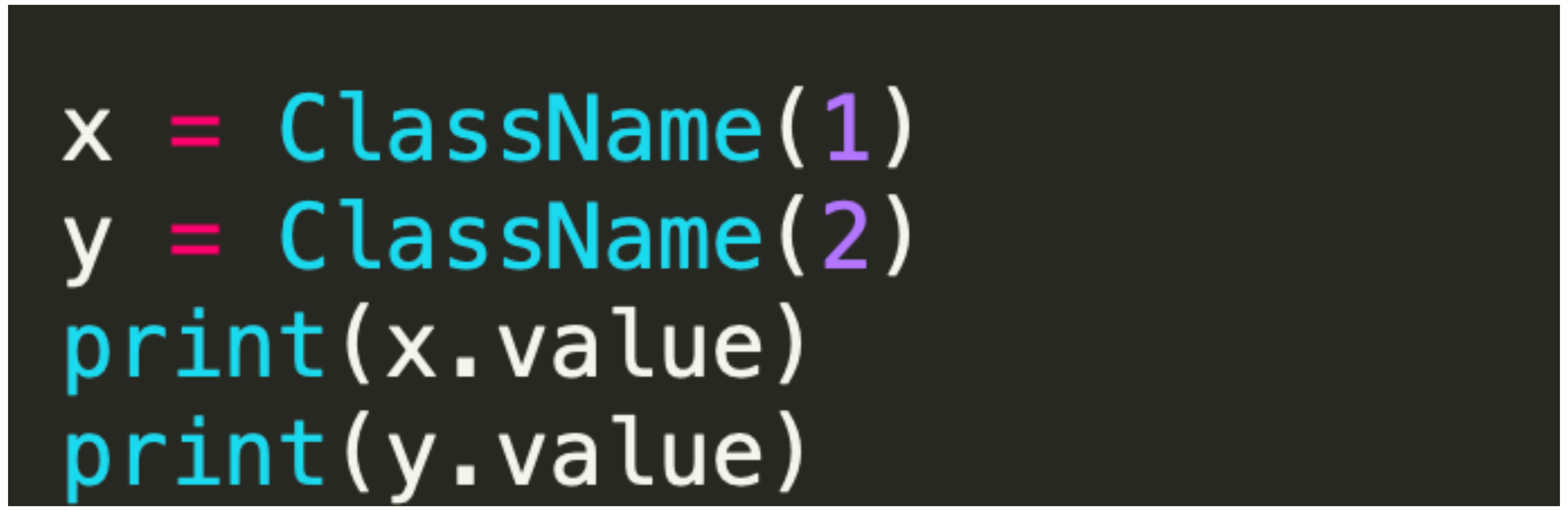
Attributes

- Attributes of an instance
    - instance variables: are for data unique to each
    - instance methods: are for manipulation of instance data
- Attributes of a class
    - class variables: are for data shared by all instances of the class
    - class methods: are for manipulation of class data

Attributes



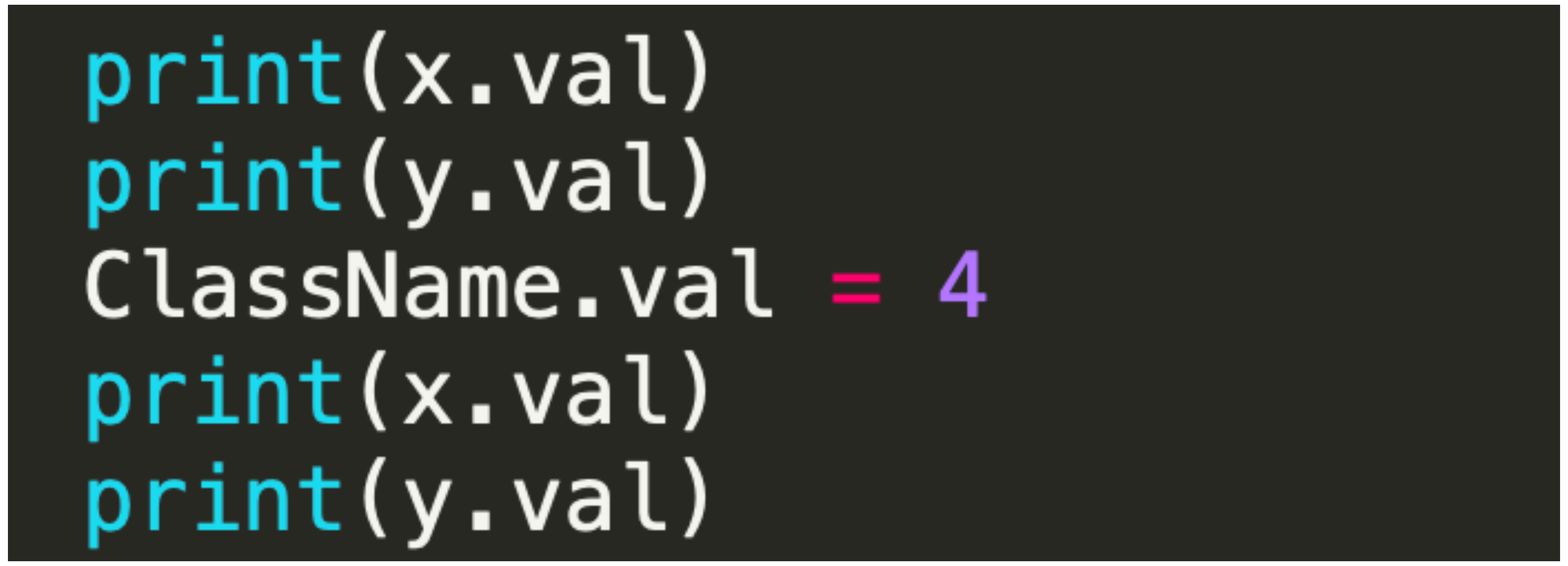
```python
class ClassName():
    val = 3
    def __init__(self, v):
        self.value = v
```



```python
x = ClassName(1)
y = ClassName(2)
print(x.value)
print(y.value)
```

Output:

```
1
2
```



```python
print(x.val)
print(y.val)
ClassName.val = 4
print(x.val)
print(y.val)
```

Output:

```
3
3
4
4
```

Access

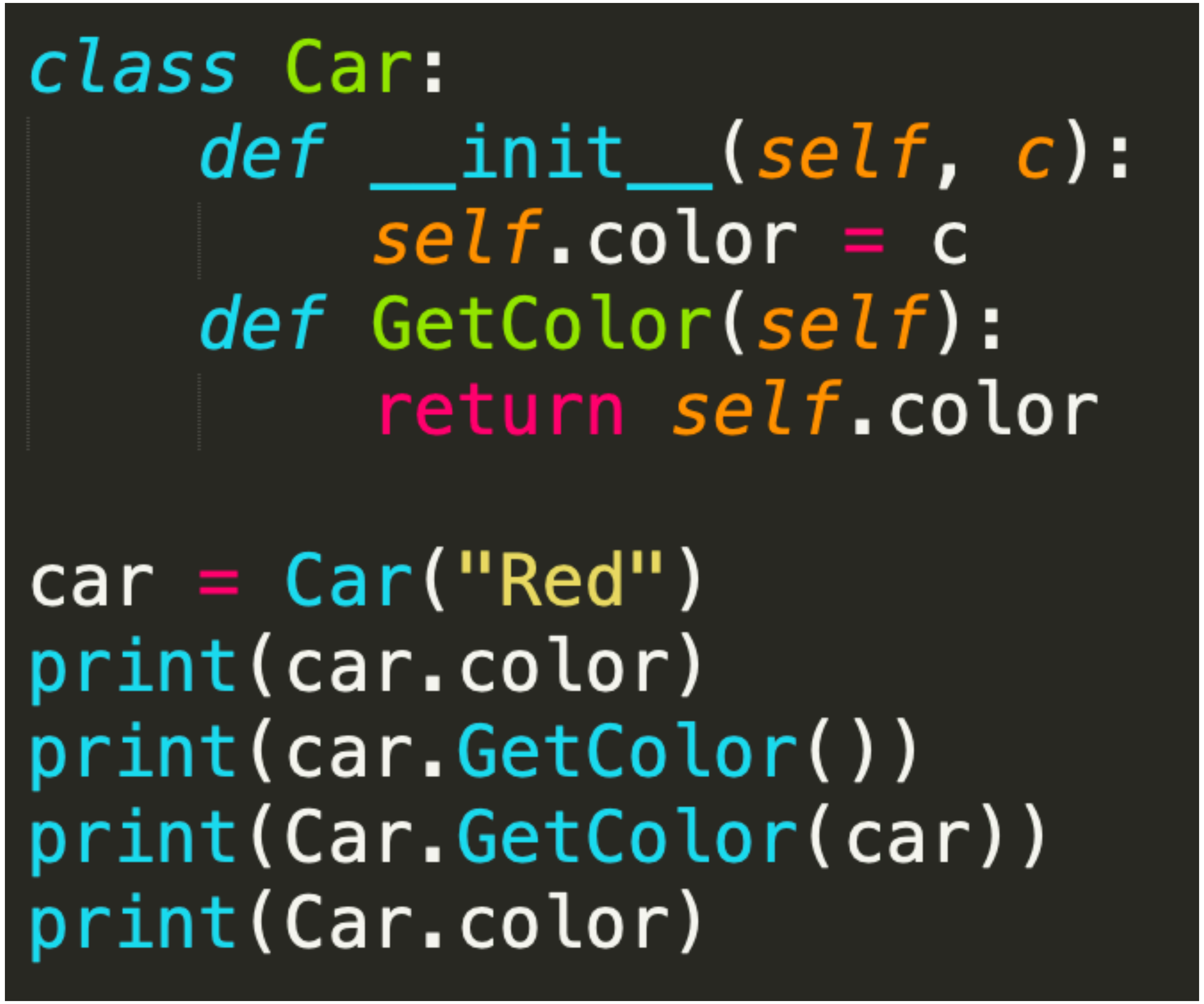
Instance variables: are accessed via object.var

Class variables: are accessed via class.var or object.var

Instance methods: are accessed via object.f(p_1,...,p_n)

Class methods: are accessed via class.f(p_1,...,p_n) or object.f(p_1,...,p_n)

Access



```
class Car:
    def __init__(self, c):
        self.color = c
    def GetColor(self):
        return self.color

car = Car("Red")
print(car.color)
print(car.GetColor())
print(Car.GetColor(car))
print(Car.color)
```

Error?

Private and Public Attributes

In Python, there is no keywords:

public, private, friend, protected

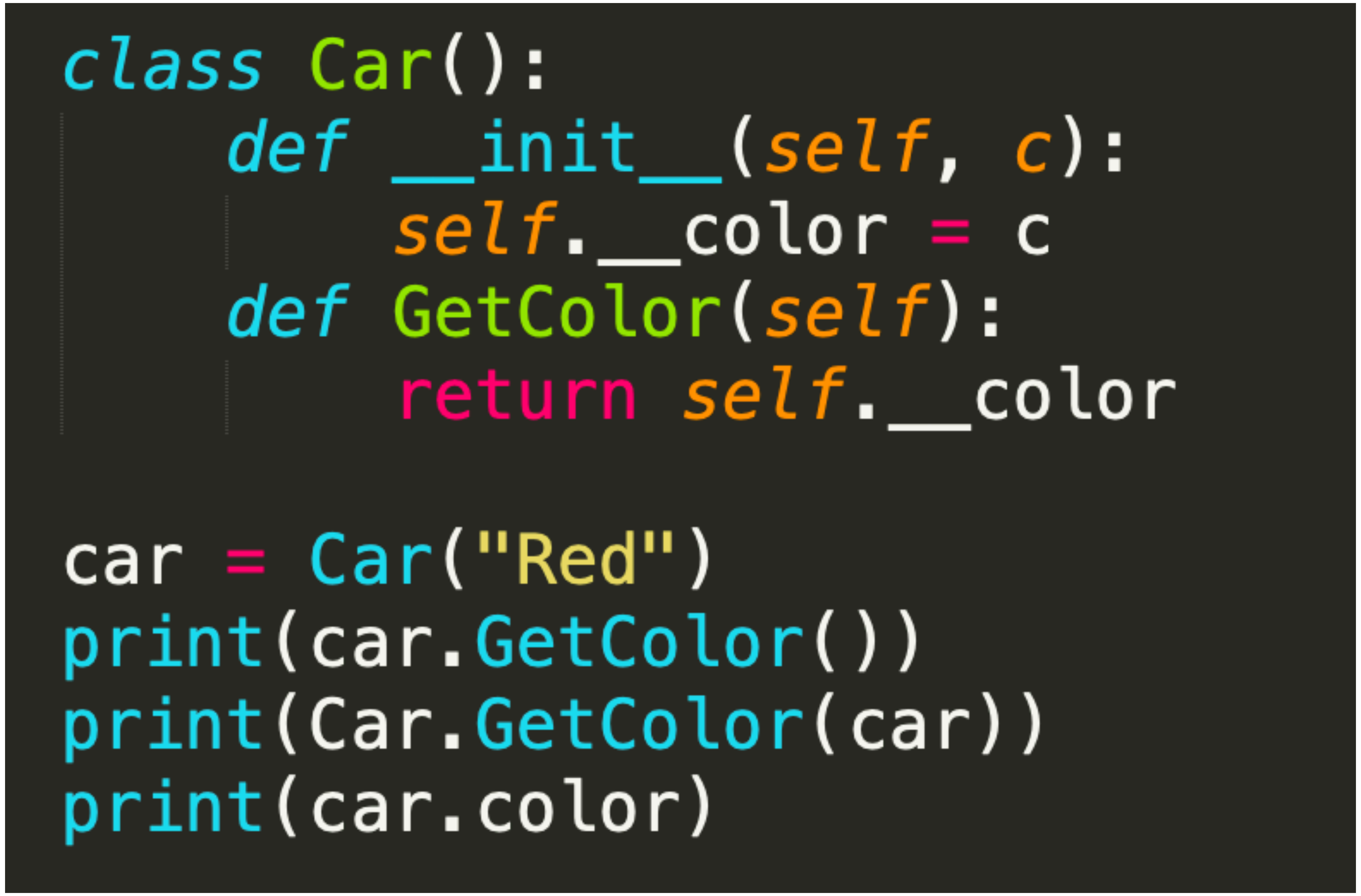
Python uses underscore to define special attributes

_xxx: denotes protected attribute xxx which cannot be imported using ‘from module import *’

__xxx__: system defined attribute xxx, e.g., __init__

__xxx: private attribute xxx, which should be accessed via instance methods, cannot be accessed via object.__xxx outside of the class, or instance methods of its subclasses (we can still access via “object._class__xxx”)

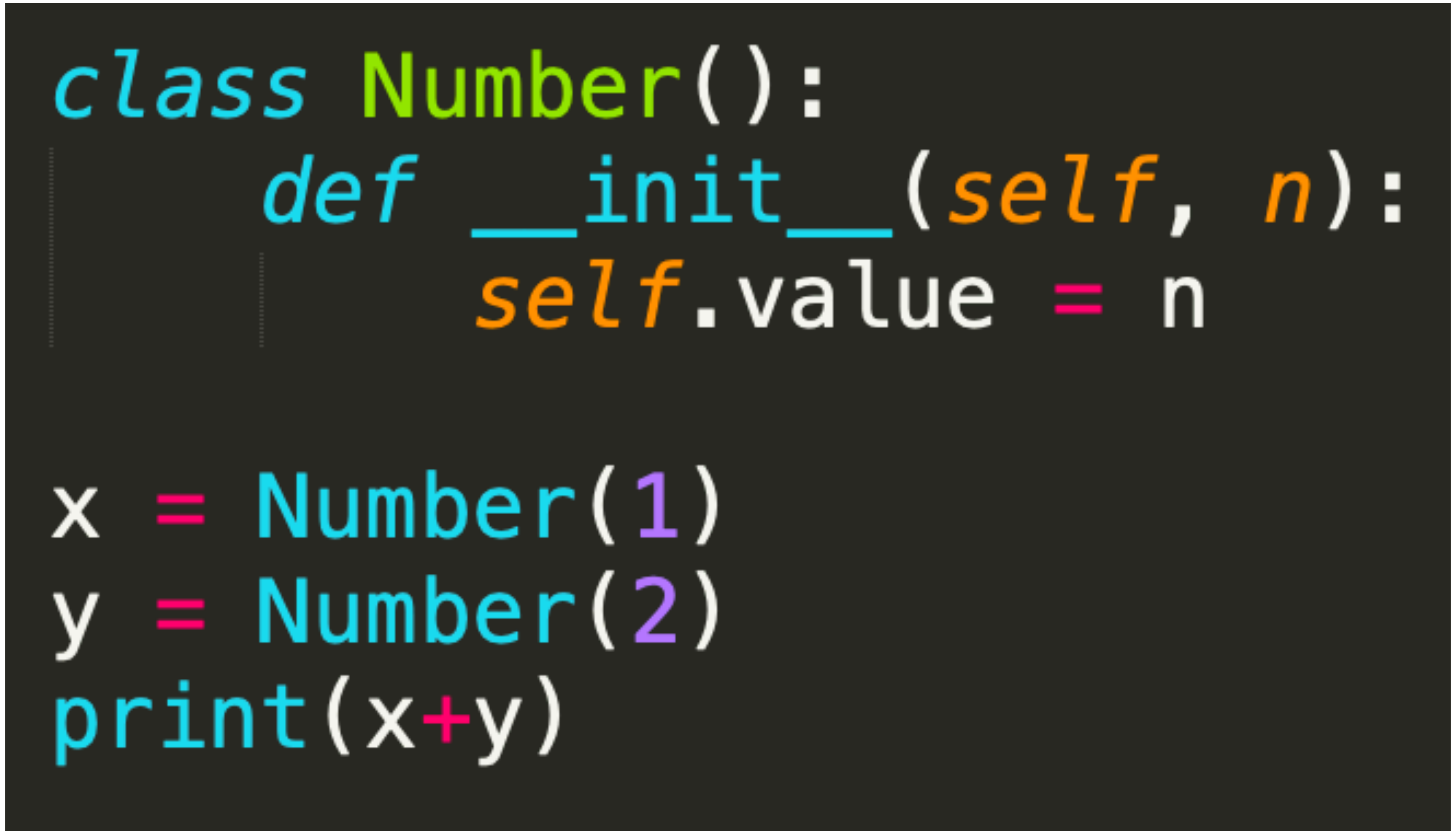
Private and Public Attributes



```python
class Car():
    def __init__(self, c):
        self.__color = c
    def GetColor(self):
        return self.__color

car = Car("Red")
print(car.GetColor())
print(Car.GetColor(car))
print(car.color)
```

Error?

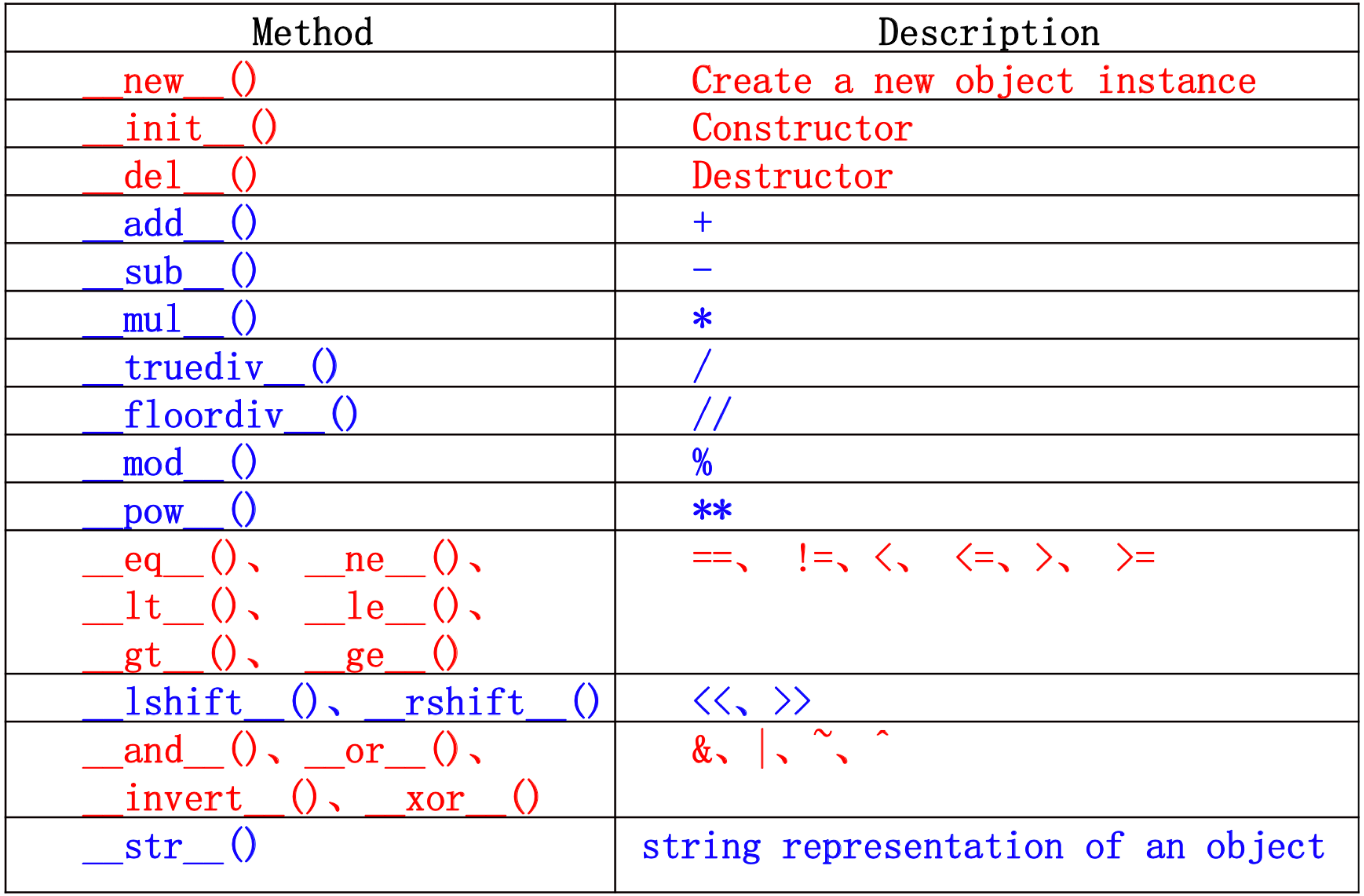
Special method names



```
class Number():
    def __init__(self, n):
        self.value = n

x = Number(1)
y = Number(2)
print(x+y)
```

Error?

Special method names

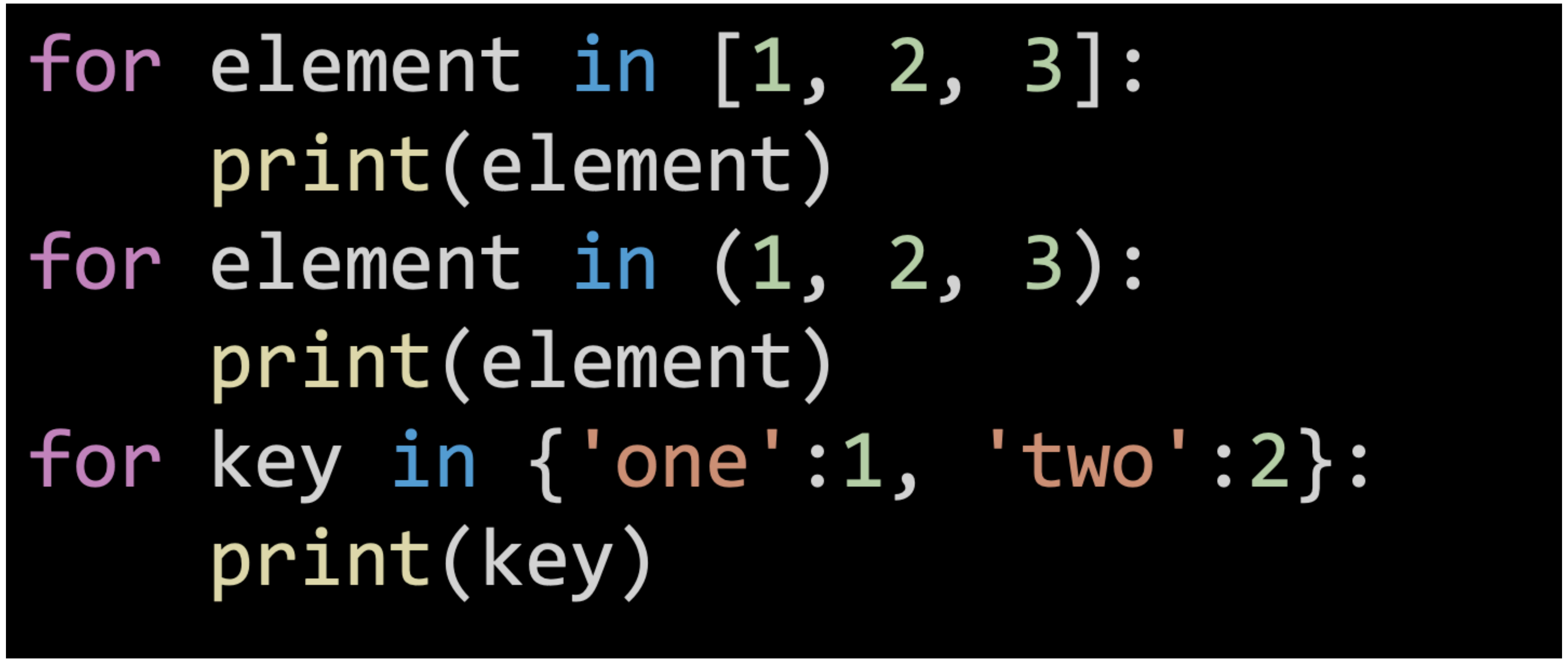


Method	Description
__new__()	Create a new object instance
__init__()	Constructor
__del__()	Destructor
__add__()	+
__sub__()	-
__mul__()	*
__truediv__()	/
__floordiv__()	//
__mod__()	%
__pow__()	**
__eq__()、 __ne__()、 __lt__()、 __le__()、 __gt__()、 __ge__()	==、 !=、<、 <=、>、 >=
__lshift__()、__rshift__()	<<、>>
__and__()、__or__()、 __invert__()、__xor__()	&、|、~、^
__str__()	string representation of an object

Iterator

Iterator

back to for loop …



```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
```

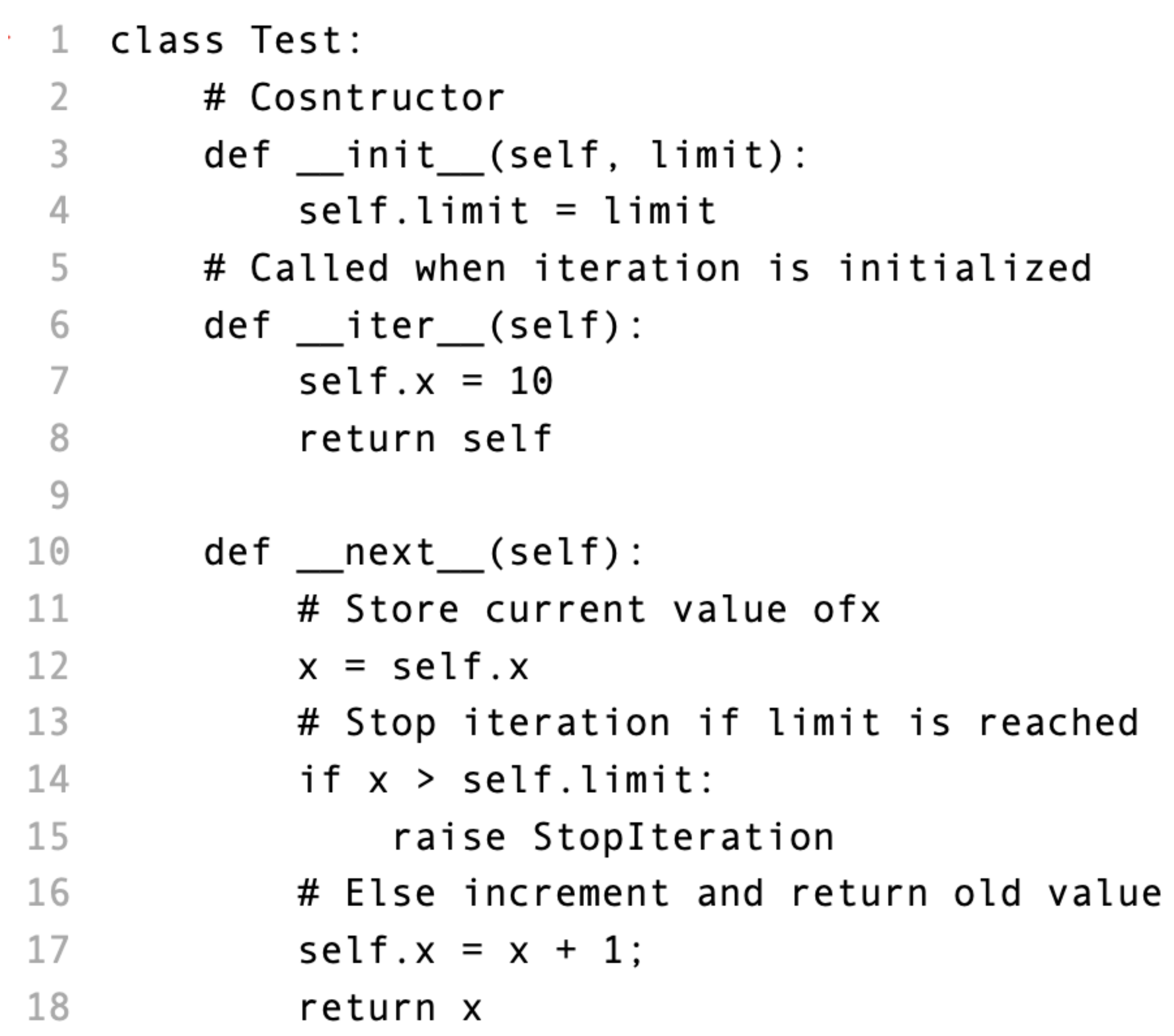
The for statement calls iter() on the container object

iter() returns an iterator object that defines __next__() which accesses elements in the container one at a time

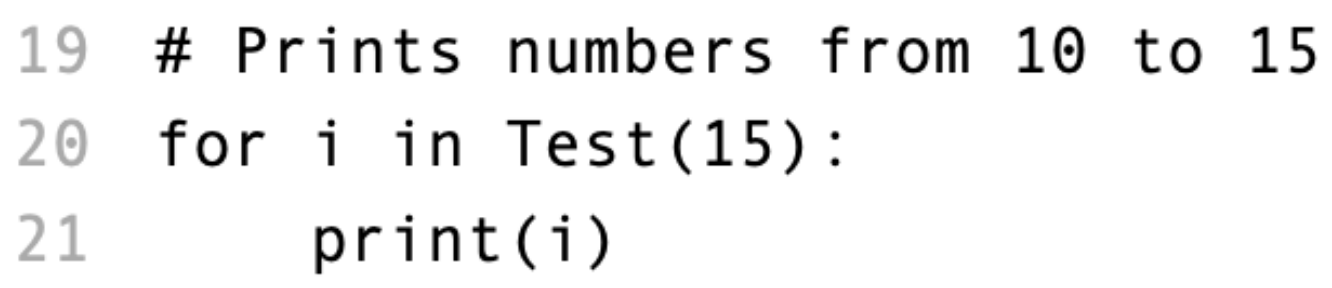
When there are no more elements, __next__() raises a StopIteration exception which tells for loop to terminate

User-defined class supporting iteration

Example



```python
class Test:
    # Cosntructor
    def __init__(self, limit):
        self.limit = limit
    # Called when iteration is initialized
    def __iter__(self):
        self.x = 10
        return self

    def __next__(self):
        # Store current value ofx
        x = self.x
        # Stop iteration if limit is reached
        if x > self.limit:
            raise StopIteration
        # Else increment and return old value
        self.x = x + 1;
        return x
```



```python
# Prints numbers from 10 to 15
for i in Test(15):
    print(i)
```

Generator

Generator

- **Generator-Function**
- **Generator-Object**

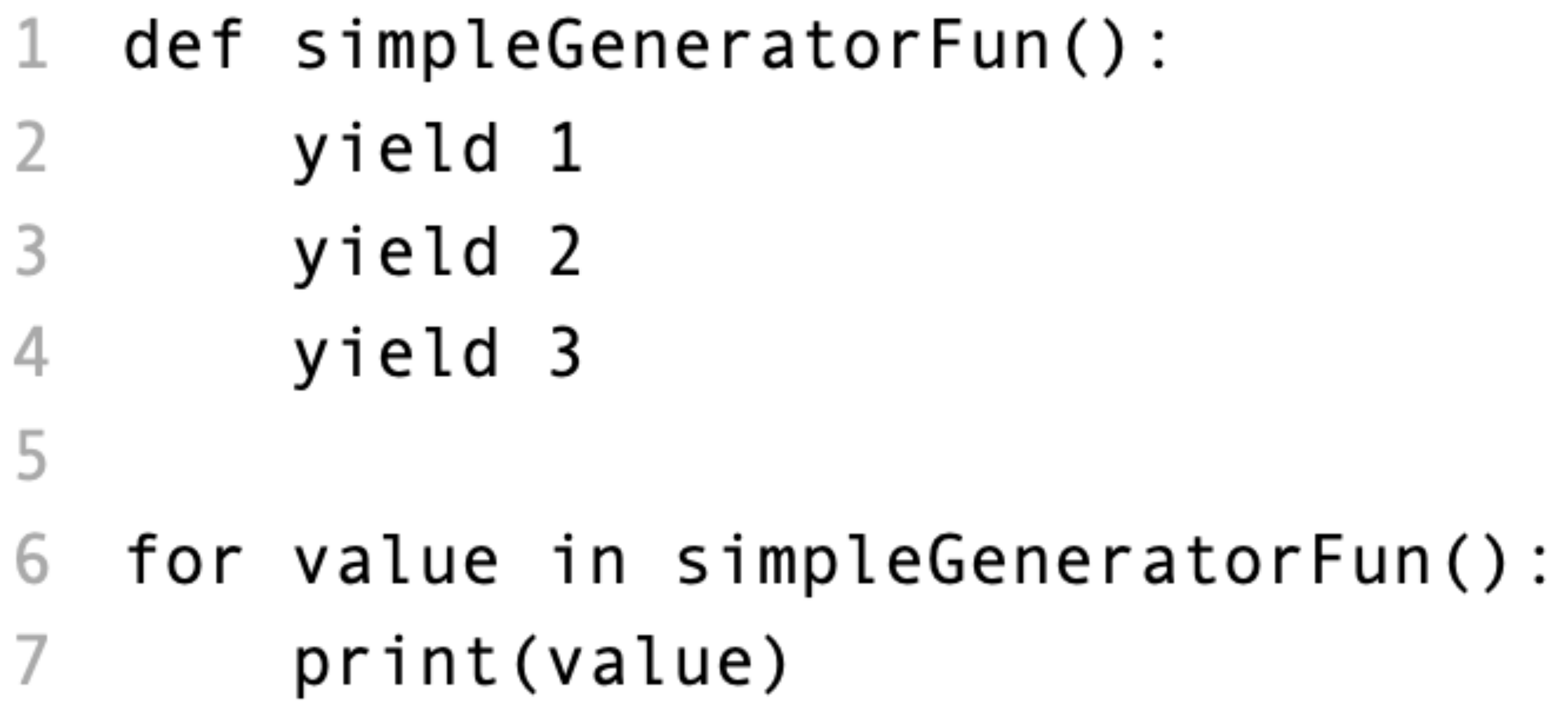
Generators are a simple and powerful tool for creating iterators

They are written like regular functions but use the yield statement whenever they want to return data

Each time __next__() is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed)

Generator-Function :

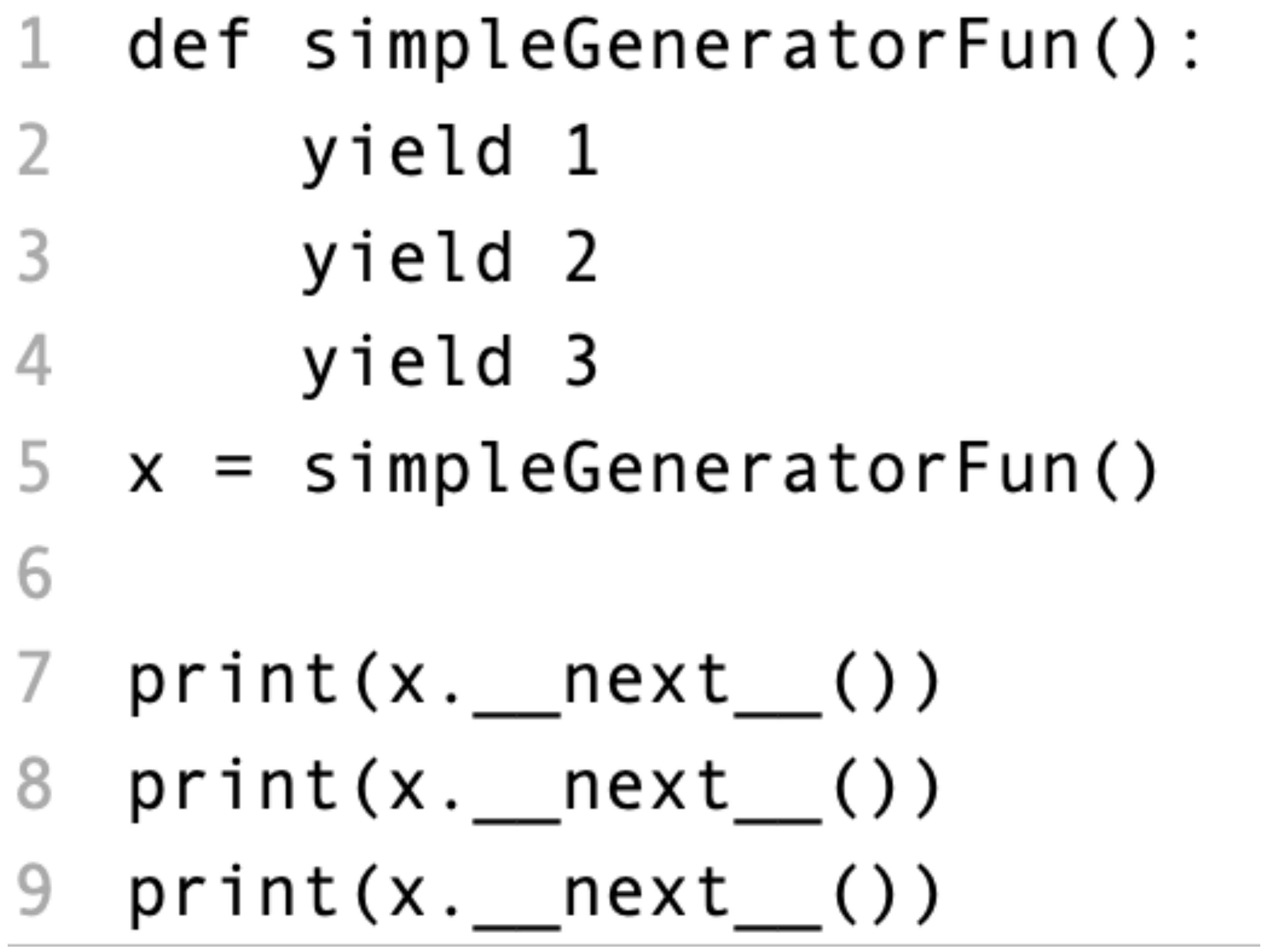
A generator-function is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return. If the body of a def contains yield, the function automatically becomes a generator function.



```python
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3

for value in simpleGeneratorFun():
    print(value)
```

Generator-Object :

Generator functions return a generator object. Generator objects are used either by calling the next method on the generator object or using the generator object in a “for in” loop.



```python
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3
x = simpleGeneratorFun()

print(x.__next__())
print(x.__next__())
print(x.__next__())
```

Task: N-Dimension Vector

Implement an N-dimention vector (for any $n \geq 1$) class:

This class should support:

1. Add(+) and Sub(-) between vectors (of the same dimension)
2. Mluti(*) and Div(/) between a vector and a scalar
3. Length of vector which is Euclidean norm of the vector
4. Pretty print of the vector

Now Code!

Testcases

```
>>> v1 = Vector([3, 4, 5])
>>> v2 = Vector([5, 6, 7])
>>> print(v1)
Vector({3},{4},{5})
>>> print(v1+v2)
Vector({8},{10},{12})
>>> print(v1-v2)
Vector({-2},{-2},{-2})
```

```
>>> print(v1*3)
Vector({9},{12},{15})
>>> print(v1/2)
Vector({1.5},{2.0},{2.5})
>>> print(v1.getLength())
7.0710678118654755
```

try __iter__() yourself

Thank you!

Wu Daqian