2023 12December 25

December 25, 2023

1 Agenda

1. Object system in Python
2. Metaclasses
3. Iterators
   - Iterator protocol
   - Adding the iterator protocol to our classes
   - Generator functions and generators

[1]:
```
class Bowl:
    pass

b = Bowl()
```

[2]:
```
type(b)
```

[2]:
```
__main__.Bowl
```

[3]:
```
type(Bowl)
```

[3]:
```
type
```

[4]:
```
type(type)
```

[4]:
```
type
```

[5]:
```
b.__class__   # this is where type info is stored
```

[5]:
```
__main__.Bowl
```

[6]:
```
b.__class__ = str
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[6], line 1
----> 1 b.__class__ = str
```

```
TypeError: __class__ assignment only supported for mutable types or ModuleType␣
↪subclasses
```

[7]:
```
Bowl.__bases__
```

[7]:
```
(object,)
```

[8]:
```
type(object)
```

[8]:
```
type
```

[9]:
```
type(str)
```

[9]:
```
type
```

[10]:
```
from collections import Counter
```

[11]:
```
type(Counter)
```

[11]:
```
type
```

[12]:
```
object.__bases__
```

[12]:
```
()
```

[13]:
```
class MyClass:
    def __init__(self, x):
        self.x = x

    # method that doesn't use self
    def hello(self):
        return f'Hello from MyClass'

m = MyClass(10)
m.hello()
```

[13]:
```
'Hello from MyClass'
```

[16]:
```
MyClass.hello()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[16], line 1
----> 1 MyClass.hello()

TypeError: MyClass.hello() missing 1 required positional argument: 'self'
```

[17]:
```
# what if I define hello without any arguments?

class MyClass:
    def __init__(self, x):
        self.x = x

    # method without any parameters
    def hello():
        return f'Hello from MyClass'

m = MyClass(10)
m.hello()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[17], line 12
      9         return f'Hello from MyClass'
     11 m = MyClass(10)
---> 12 m.hello()

TypeError: MyClass.hello() takes 0 positional arguments but 1 was given
```

[18]:
```
MyClass.hello()
```

[18]:
```
'Hello from MyClass'
```

[19]:
```
# how can I have a method that works from both the instance and class, but␣
 ↪doesn't
# require passing an instance?

class MyClass:
    def __init__(self, x):
        self.x = x

    @staticmethod
    def hello():
        return f'Hello from MyClass'

m = MyClass(10)
m.hello()
```

[19]:
```
'Hello from MyClass'
```

[20]:
```
MyClass.hello()
```

[20]:
```
'Hello from MyClass'
```

[21]:
```python
# @classmethod


class MyClass:
    def __init__(self, x):
        self.x = x

    @classmethod
    def hello(cls):    # class methods get the class as an argument
        return f'Hello from {cls}'

m = MyClass(10)
m.hello()
```

[21]:
```
"Hello from <class '__main__.MyClass'>"
```

[22]:
```python
MyClass.hello()
```

[22]:
```
"Hello from <class '__main__.MyClass'>"
```

[23]:
```python
dir(MyClass)
```

[23]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
```

```
 '__subclasshook__',
 '__weakref__',
 'hello']
```

[24]:
```
vars(MyClass)
```

[24]:
```
mappingproxy({'__module__': '__main__',
              '__init__': <function __main__.MyClass.__init__(self, x)>,
              'hello': <classmethod(<function MyClass.hello at 0x107b4c2c0>)>,
              '__dict__': <attribute '__dict__' of 'MyClass' objects>,
              '__weakref__': <attribute '__weakref__' of 'MyClass' objects>,
              '__doc__': None})
```

[28]:
```
# standard way to do this
class ThisType:
    x = 100

# I could also have said:
t = type('ThisType', (object,), {'x':100})
```

[29]:
```
type(t)
```

[29]:
```
type
```

[30]:
```
dir(t)
```

[30]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
```

```
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'x']
```

[31]:
```
t.x
```

[31]:
```
100
```

[32]:
```
class u(t):
    pass
```

[33]:
```
u.x
```

[33]:
```
100
```

[36]:
```
class MyMeta(type):       # MyMeta inherits from type, and can thus be a␣
 ↪metaclass
    def __new__(cls, name, bases, attributes):
        return super().__new__(cls, name, bases, attributes)

class MyClass(metaclass=MyMeta):
    pass

m = MyClass()
print(m)
```

```
<__main__.MyClass object at 0x107bb7c10>
```

[37]:
```
type(MyClass)
```

[37]:
```
__main__.MyMeta
```

[38]:
```
isinstance(MyMeta, type)
```

[38]:
```
True
```

[ ]:
```
class MyMeta(type):       # MyMeta inherits from type, and can thus be a␣
 ↪metaclass
    def __new__(cls, name, bases, attributes):
        return super().__new__(cls, name, bases, attributes)

class MyClass(metaclass=MyMeta):
    pass
```

```
m = MyClass()
print(m)
```

`MyClass`:

- `type` is `MyMeta`
- inherits from `object`

Thus: - If I ask for MyClass.z, Python will look: - Instance: `MyClass` - Class: `MyMeta` - Parent/`object`: `object`

- If I ask for m.z, Python will look:
  - Instance: `m`
  - Class: `MyClass`
  - Parent/`object`: `object`

[43]:
```
# let's add a new attribute

class MyMeta(type):       # MyMeta inherits from type, and can thus be a␣
 ↪metaclass
    def __new__(cls, name, bases, attributes):

        attributes['x'] = 100
        attributes['y'] = [10, 20, 30]

        # add a method, while we're at it
        def hello(self):
            return f'Hello!'
        attributes['hello'] = hello

        return super().__new__(cls, name, bases, attributes)

class MyClass(metaclass=MyMeta):
    pass

m = MyClass()  # this runs MyMeta.__new__, which adds to attributes, then runs␣
 ↪object.__new__ and object.__init__
print(m)
```

```
<__main__.MyClass object at 0x107c0b3d0>
```

[44]:
```
m.x   # does m have x? No. Does m's class (MyClass) have x? Yes, thanks to␣
 ↪MyMeta.__new__
```

[44]:
```
100
```

[45]:
```
m.y    # does m have y? No. Does m's class (MyClass) have y? Yes, thanks to␣
 ↪MyMeta.__new__
```

[45]:
```
[10, 20, 30]
```

[ ]:

[46]:
```
m.hello()
```

[46]:
```
'Hello!'
```

[47]:
```
vars(MyClass)
```

[47]:
```
mappingproxy({'__module__': '__main__',
              'x': 100,
              'y': [10, 20, 30],
              'hello': <function __main__.MyMeta.__new__.<locals>.hello(self)>,
              '__dict__': <attribute '__dict__' of 'MyClass' objects>,
              '__weakref__': <attribute '__weakref__' of 'MyClass' objects>,
              '__doc__': None})
```

My talk about decorators from PyCon US 2019:

https://www.youtube.com/watch?v=MjHpMCIvwsY

2 Properties and descriptors

When I call a method on an instance, the call is rewritten:

```
s = 'abcd'
s.lower()    # this is rewritten to be str.lower(s)
```

Who is doing this rewriting?

A descriptor is an object that:

- we define as a class attribute
- we access via the instance

When we do this: - If we ask for its value, we don't get the class attribute's value back. Rather, we run `__get__` on the object, and return its value - If we assign to it, we don't set the attribute's value. Rather, we run `__set__` on the object

We see this every day, when we call a method!

- Methods are defined on the class
- We call them via the instance
- The descriptor sees the call via the instance, and rewrites the call to make the instance `self`

Properties are easy-to-create descriptors, that look like values but act like methods. That's because they are actually methods!

[49]:
```
class MyClass:
    def __init__(self):
        self._value = None

    @property
```

```
    def value(self):
        print('Now in value getter!')
        return self._value

m = MyClass()
m.value     # ask via the instance, but value is defined on the class
```

```
Now in value getter!
```

[50]:
```
class MyClass:
    def __init__(self):
        self._value = None

    @property      # this decorator means: the following method is a getter
    def value(self):
        print('Now in value getter!')
        return self._value

    @value.setter  # this decorator means: the following method is a setter for␣
 ↪value
    def value(self, new_value):
        print(f'Assigning {new_value} to value!')
        self._value = new_value

m = MyClass()
print(m.value)
m.value = 12345
print(m.value)
```

```
Now in value getter!
None
Assigning 12345 to value!
Now in value getter!
12345
```

[53]:
```
class MyClass:
    def __init__(self, x):
        self.x = x

    def x2(self):
        return self.x * 2

    # if the attribute doesn't exist, then this method is called with the␣
 ↪requested name
    def __getattr__(self, name):
        return f'Got __getattr__ with {name}'

m = MyClass(10)
```

[54]:
```
m.x
```

[54]:
```
10
```

[55]:
```
m.y
```

[55]:
```
'Got __getattr__ with y'
```

[56]:
```
m.__str__()
```

[56]:
```
'<__main__.MyClass object at 0x107c54d50>'
```

[57]:
```
dir(MyClass)
```

[57]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattr__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'x2']
```

[58]:
```
dir(m)
```

[58]:
```
['__class__',
 '__delattr__',
```

```
'__dict__',
'__dir__',
'__doc__',
'__eq__',
'__format__',
'__ge__',
'__getattr__',
'__getattribute__',
'__getstate__',
'__gt__',
'__hash__',
'__init__',
'__init_subclass__',
'__le__',
'__lt__',
'__module__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'__weakref__',
'x',
'x2']
```

[59]:
```
class MyClass:
    pass

m = MyClass()
len(m)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[59], line 5
      2     pass
      4 m = MyClass()
----> 5 len(m)

TypeError: object of type 'MyClass' has no len()
```

[60]:
```
dir(m)
```

[60]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__']
```

[61]:
```
class MyClass:
    def __len__(self):
        return 5

m = MyClass()
len(m)
```

[61]:
```
5
```

[62]:
```
dir(m)
```

[62]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
```

```
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__len__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__']
```

[63]:
```
class MyClass:
    x = 100     # x is a class attribute, its value is an instance of int, aka␣
 ↪100

m = MyClass()
m.x
```

[63]: 100

[64]:
```
class MyDescriptor:
    def __get__(self, instance, owner):
        print(f'In MyDescriptor.__get__')
        return 12345

class MyClass:
    x = MyDescriptor()   # x is a class attribute, its value is instance of␣
 ↪MyDescriptor

m = MyClass()
m.x
```

```
In MyDescriptor.__get__
```

[64]: 12345

[65]:
```
class MyClass:
    def hello(self):
        return f'Hello!'
```

[66]:
```
MyClass.hello
```

[66]:
```
<function __main__.MyClass.hello(self)>
```

[67]:
```
m = MyClass()
m.hello
```

[67]:
```
<bound method MyClass.hello of <__main__.MyClass object at 0x107c2d4d0>>
```

[68]:
```
m.hello()
```

[68]:
```
'Hello!'
```

3 Next up:

1. Iterators
2. Iterators as classes
3. Generators

Resume at :25

[3]:
```
class MyDescriptor:
    def __init__(self):
        print(f'In MyDescriptor.__init__')
    def __get__(self, instance, owner):
        print(f'In MyDescriptor.__get__')
        return 12345
    def __del__(self):
        print(f'Now deleting instance of MyDescriptor')

class MyClass:
    x = MyDescriptor()    # x is a class attribute, its value is instance of␣
 ↪MyDescriptor

m = MyClass()
print(m.x)
print(m.x * 2)
```

```
In MyDescriptor.__init__
In MyDescriptor.__get__
12345
In MyDescriptor.__get__
24690
```

[5]:
```
del(m)
```

[6]:
```
del(MyClass)
```

[7]:
```
del(MyDescriptor)
```

[8]:
```
import gc
```

[9]:
```
gc.get_referrers(MyDescriptor)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[9], line 1
----> 1 gc.get_referrers(MyDescriptor)

NameError: name 'MyDescriptor' is not defined
```

[12]:
```
# context manager

class MyCM:
    def __init__(self, x):
        print(f'In MyCM.__init__, {x=}')
        self.x = x

    def __enter__(self):
        print(f'In MyCM.__enter__')
        return self

    def __exit__(self, *args):
        print(f'In MyCM.__exit__, {args=}')
        return True

m = MyCM(10)
print(m.x)
```

```
In MyCM.__init__, x=10
10
```

[13]:
```
with MyCM(10) as m:
    # __enter__
    print('Inside')
    # __exit__
```

```
In MyCM.__init__, x=10
In MyCM.__enter__
Inside
In MyCM.__exit__, args=(None, None, None)
```

4 Iteration!

How does a `for` loop work in Python?

1. `for` turns to the object at the end of the line, and asks if it's iterable.
   - If not, then we exit with a `TypeError`
2. `for` asks for the next item that the object has to offer
   - If there aren't any more, the loop ends
3. The next item from the object is assigned to our variable, and the loop body executes.
4. Goto 2

[14]:
```
s = 'abcd'

for one_character in s:
    print(one_character)
```

```
a
b
c
d
```

5 In more detail:

1. `for` turns to the object at the end of the line, and asks if it's iterable, using the builtin `iter` function. We get back an iterator object.
   - If not, then we exit with a `TypeError`
2. `for` asks the iterator (using `next`) for the next item that the object has to offer
   - If there aren't any more, we get `StopIteration`
3. The next item from the object is assigned to our variable, and the loop body executes.
4. Goto 2

[15]:
```
iter(s)
```

[15]:
```
<str_ascii_iterator at 0x10ced5de0>
```

[16]:
```
iter([10, 20, 30])
```

[16]:
```
<list_iterator at 0x10ced65c0>
```

[17]:
```
iter(5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[17], line 1
----> 1 iter(5)

TypeError: 'int' object is not iterable
```

[18]:
```
i = iter(s)

next(i)
```

[18]:
```
'a'
```

[19]:
```
next(i)
```

[19]:
```
'b'
```

[20]:
```
next(i)
```

[20]:
```
'c'
```

[21]:
```
next(i)
```

[21]:
```
'd'
```

[22]:
```
next(i)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[22], line 1
----> 1 next(i)

StopIteration:
```

6 How can I add this to my classes?

1. I need to know how to respond to `iter` – using a method called `__iter__`, which returns an iterator object, one that knows how to respond to `next`
2. I need to know how to respond to `next` – using a method called `__next__`, which does one of two things:
   - returns the next object
   - raises `StopIteration`

[23]:
```
class MyIterator:
    def __init__(self, data):
        print(f'\tIn __init__, {data=}')
        self.data = data
        self.index = 0

    def __iter__(self):
        print(f'\tIn __iter__, {vars(self)=}')
        return self
```

```
    def __next__(self):
        print(f'\tIn __next__, {vars(self)=}')
        if self.index >= len(self.data):
            print(f'\t\tRaising StopIteration')
            raise StopIteration

        value = self.data[self.index]
        self.index += 1
        print(f'\t\tReturning {value=}')
        return value

m = MyIterator('abcd')
for one_item in m:
    print(one_item)
```

```
        In __init__, data='abcd'
        In __iter__, vars(self)={'data': 'abcd', 'index': 0}
        In __next__, vars(self)={'data': 'abcd', 'index': 0}
                Returning value='a'
a
        In __next__, vars(self)={'data': 'abcd', 'index': 1}
                Returning value='b'
b
        In __next__, vars(self)={'data': 'abcd', 'index': 2}
                Returning value='c'
c
        In __next__, vars(self)={'data': 'abcd', 'index': 3}
                Returning value='d'
d
        In __next__, vars(self)={'data': 'abcd', 'index': 4}
                Raising StopIteration
```

[24]:
```
len(m)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[24], line 1
----> 1 len(m)

TypeError: object of type 'MyIterator' has no len()
```

7 Exercise: Circle

1. Implement `Circle`, a class that takes two arguments:
   - First, data that's a sequence (string, list, tuple)

- Second, the number of values we want to get from it when we iterate (`maxtimes`)

2. If `maxtimes` is smaller than the length of our sequence, then iterating over our object should give us `maxtimes` values.
3. If `maxtimes` is larger than the length of our sequence, then we should go through the values, and then go back to the start as many times as needed to get `maxtimes` values.

Example:

```
c = Circle('abcd', 9)
for one_item in c:
    print(one_item)    # a b c d a b c d a
```

[26]:
```
class Circle:
    def __init__(self, data, maxtimes):
        self.data = data
        self.maxtimes = maxtimes
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index >= self.maxtimes:
            raise StopIteration

        value = self.data[self.index % len(self.data)]
        self.index += 1
        return value

c = Circle('abcd', 5)
for one_item in c:
    print(one_item)
```

```
a
b
c
d
a
```

[27]:
```
s = 'abcd'

print('** first time **')
for one_item in s:
    print(one_item)

print('** second time **')
for one_item in s:
    print(one_item)
```

```
** first time **
a
b
c
d
** second time **
a
b
c
d
```

[28]:
```
c = Circle('abcd', 5)

print('** first time **')
for one_item in c:
    print(one_item)

print('** second time **')
for one_item in c:
    print(one_item)
```

```
** first time **
a
b
c
d
a
** second time **
```

[29]:
```
class CircleIterator:
    def __init__(self, data, maxtimes):
        self.data = data
        self.maxtimes = maxtimes
        self.index = 0

    def __next__(self):
        if self.index >= self.maxtimes:
            raise StopIteration

        value = self.data[self.index % len(self.data)]
        self.index += 1
        return value


class Circle:
    def __init__(self, data, maxtimes):
        self.data = data
        self.maxtimes = maxtimes
```

```
    def __iter__(self):
        return CircleIterator(self.data, self.maxtimes)

c = Circle('abcd', 5)

print('** first time **')
for one_item in c:
    print(one_item)

print('** second time **')
for one_item in c:
    print(one_item)
```

```
** first time **
a
b
c
d
a
** second time **
a
b
c
d
a
```

[30]:
```
c = Circle('abcd', 5)

i1 = iter(c)
i2 = iter(c)
```

[31]:
```
next(i1)
```

[31]: 'a'

[32]:
```
next(i1)
```

[32]: 'b'

[33]:
```
next(i1)
```

[33]: 'c'

[34]:
```
next(i2)
```

[34]: 'a'

8 Exercise: OnlyVowels

1. Define `OnlyVowels`, a class that takes a single string arguments.
2. If I iterate with a `for` loop over an instance of `OnlyVowels`, I'll get (one by one) the vowels (a, e, i, o, u) from there.
3. Implement this with two classes, rather than one.

[35]:
```
class OnlyVowels:
    def __init__(self, data):
        self.data = data
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index >= len(self.data):
            raise StopIteration

        value = self.data[self.index]
        self.index += 1
        if value in 'aeiou':
            return value
        return self.__next__()

o = OnlyVowels('this is a test')
for one_item in o:
    print(one_item)
```

```
i
i
a
e
```

[38]:
```
# two-class version

class OnlyVowelsIterator:
    def __init__(self, ov):
        self.only_vowels = ov
        self.index = 0

    def __next__(self):
        if self.index >= len(self.only_vowels.data):
            raise StopIteration

        value = self.only_vowels.data[self.index]
        self.index += 1
```

```
        if value in 'aeiou':
            return value
        return self.__next__()

class OnlyVowels:
    def __init__(self, data):
        self.data = data

    def __iter__(self):
        return OnlyVowelsIterator(self)


o = OnlyVowels('this is a test')

print('** first time **')
for one_item in o:
    print(one_item)

print('** second time **')
for one_item in o:
    print(one_item)
```

```
** first time **
i
i
a
e
** second time **
i
i
a
e
```

[39]:
```
list(o)
```

[39]:
```
['i', 'i', 'a', 'e']
```

[41]:
```
{one_item
  for one_item in o}
```

[41]:
```
{'a', 'e', 'i'}
```

9 Generators

[42]:
```
def myfunc():
    return 1
    return 2
```

```
    return 3
```

[43]:
```
myfunc()
```

[43]:
```
1
```

[44]:
```
import dis
```

[45]:
```
dis.dis(myfunc)
```

```
  1           0 RESUME                   0

  2           2 LOAD_CONST               1 (1)
              4 RETURN_VALUE
```

[46]:
```
def myfunc():
    yield 1
    yield 2
    yield 3
```

[48]:
```
# I get a generator object back!
# generators implement the iterator protocol
myfunc()
```

[48]:
```
<generator object myfunc at 0x10ce33e20>
```

[49]:
```
g = myfunc()
```

[50]:
```
next(g)
```

[50]:
```
1
```

[51]:
```
next(g)
```

[51]:
```
2
```

[52]:
```
next(g)
```

[52]:
```
3
```

[53]:
```
next(g)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[53], line 1
----> 1 next(g)
```

```
StopIteration:
```

[54]:
```
# generator functions have "yield" in there somewhere
# running a generator function returns a generator object
```

[55]:
```
def fib():
    first = 0
    second = 1

    while True:
        yield first
        first, second = second, first+second
```

[56]:
```
g = fib()    # create a generator for Fibonacci numbers
```

[57]:
```
for one_item in g:
    if one_item > 100_000_000:
        break

    print(one_item, end=' ')
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711
28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578
5702887 9227465 14930352 24157817 39088169 63245986
```

[58]:
```
dir(g)
```

[58]:
```
['__class__',
 '__del__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__lt__',
 '__name__',
 '__ne__',
```

```
 '__new__',
 '__next__',
 '__qualname__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'close',
 'gi_code',
 'gi_frame',
 'gi_running',
 'gi_suspended',
 'gi_yieldfrom',
 'send',
 'throw']
```

[59]:
```
dir(g.gi_frame)
```

[59]:
```
['__class__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'clear',
 'f_back',
```

```
 'f_builtins',
 'f_code',
 'f_globals',
 'f_lasti',
 'f_lineno',
 'f_locals',
 'f_trace',
 'f_trace_lines',
 'f_trace_opcodes']
```

[60]:
```
g.gi_frame.f_locals
```

[60]:
```
{'first': 102334155, 'second': 165580141}
```

[61]:
```
g.gi_frame.f_lineno
```

[61]:
```
6
```

[62]:
```
next(g)
```

[62]:
```
165580141
```

[63]:
```
g.gi_frame.f_locals
```

[63]:
```
{'first': 165580141, 'second': 267914296}
```

10 Next time:

- Generator functions
- Generator comprehensions
- Concurrency
-

[ ]: