ASYNCHRONOUS AND EVENT-BASED PROGRAMMING

Dag Brattli
FAST Engineering- Office 365 Services
Microsoft

Asynchronous and Event-Based Programming

- An increasingly popular style of programming
  - *Javascript, Node.js, Async/await (C#, Python, etc)*
- Considerably improves the responsiveness for many application
- The foundation for Reactive Programming

"If you do nothing, you can scale infinitely." - Hanselman's rule of scale

Collections

A something of something!

- In Python a collection is defined as a container
- Containers may be sequences, sets and mappings.
- A collection is an abstraction of something that:
  - *Has a size*
  - *May contain something*
  - *Be iterable*
- Collections may be defined, iterated and generated

```
# Collections
xs = [1, 2, 3, 4]

ys = {'a': 1, 'b': 2, 'c': 3}

zs = (1, 2, 3, "a")
```

Iterables

```
from abc import ABCMeta, abstractmethod

class Iterable(metaclass=ABCMeta):
    @abstractmethod
    def __iter__(self):
        while False:
            yield None
```

- Collections may be iterable
- An iterable is an object that is capable of returning its members one at a time.
- Python has the concept of iteration over collections using what's called the iterator protocol
- An iterable is also called a lazy collection
- Iterables also makes it possible to create collections that represents infinite sequences
- Before Python 2.3 there were no iterators, only indexing

```
# An Iterable

for x in range(10):
    print(x)
```

Iterators

- An iterator is an object representing a “stream” of items
- An iterable may return several iterators for the same collection
- This is because the collection may be “viewed” simultaneously by many at the same time
- Each iterator knows it’s current position within in the collection being iterated

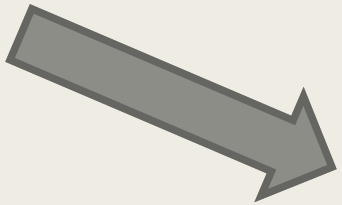
```
class Iterator(Iterable):
    @abstractmethod
    def __next__(self):
        # Return the next item from the iterator
        raise StopIteration

    def __iter__(self):
        return self
```

Show me the code ...

But where is my iterator?

- We usually never have to deal with iterators when programming in Python
- This is because they are hidden by various programming constructs within the language

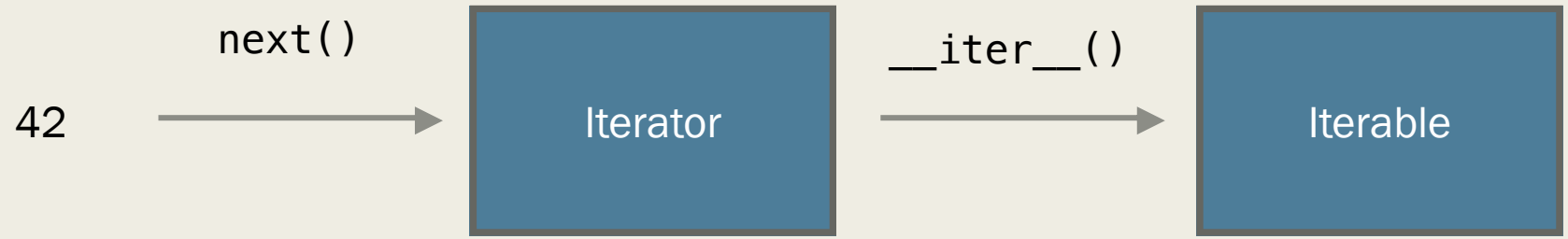
```
for x in [1, 2, 3]:
    print(x)
```



```
iterator = iter([1, 2, 3])
while True:
    try:
        x = next(iterator)
    except StopIteration:
        break
    print(x)
```

Summery

- A **getter** is a function that takes nothing and returns a value
- So an iterable is nothing more than a getter that returns an iterator
- An iterator is nothing more than a getter that returns a value
- An iterable is a **getter getter**



Generators

- A generator is a special Python function that when called produces an iterable sequence
- A generator is a special kind of iterator
- Any function or method that uses the **yield** keyword is by definition a generator
- May produce infinite sequences

```
def gen123():
    yield 1
    yield 2
    yield 3

for n in gen123():
    print(n)
```

```
def infinite():
    """A generator representing an infinite sequence."""
    i = 0
    while True:
        yield i
        i += 1
```

Yield from ...

- Python 3.3 and PEP-380 also introduced the yield from expression to delegate work to a sub-generator.
- This makes it possible for us to chain multiple generators:

```
def gen1():
    yield 1
    yield 2

def gen2()
    yield 3

def gen3():
    # Delegates the work to the two sub-generators
    yield from gen1()
    yield from gen2()

for x in gen3():
   print(x)
```

Enhanced generators

- Python 2.5 and PEP-342 introduced several new features for generators
- Previously generators could only produce values, but now it's also possible to send values and even exceptions into the generator
- Add 3 new methods,
  - *send(value)*
  - *throw(ex)*
  - *close()*
- https://github.com/python/cpython/blob/master/Lib/_collections_abc.py

Show me some code ...

```
def gen():
    # You can receive sent values from yield
    x = yield 1
    print("generator received: %d" % x)

    try:
        # You can receive send exceptions
        x = yield 2
    except Exception as e:
        print("generator received exception: %s" % e)

    # You can yield an expression
    x = 40 + (yield 3)
    print(x)
```

But, does this have anything to do with asynchronous and event-based programming?

Synchronous programming

- Synchronous programming is programming with synchronous functions
- A synchronous function produces its result as the return value of the function
- We say the function blocks while waiting for the result
  - *Usually this is very fast*
  - *But sometimes very slow, functions may block for a very long time*
- What's the problem?

```
result = input()  # may block forever
print(result)
```

Here be dragons!

- Multithreading. It sounded like a good idea
- In order to do more than one thing at the same time when using synchronous programming we have to use multiple threads and this is called multi-threaded programming
- Multi-threaded programs can be very hard to write correctly, and programs that are very hard to write correctly are extremely hard to debug

> Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it? --- Brian W. Kernighan

- Threads also consume system resources, so a multi-threaded web server may have a hard time serving hundreds of thousands of concurrent requests and connections, even if the web server spends most of its time just waiting for disk I/O
- Multithreaded programs are almost impossible to unit-test. How do you sleep at night?

Asynchronous programming

- Asynchronous programming is programming with asynchronous functions
- An asynchronous function returns before the result is ready
- To get the result we need to give it a callback handler that will call back with the result when it's ready

```
def callback(result):
    """Called later when result is ready"""
    print(result)

input_async(callback)
# Lets do something else while waiting for input
```

Asynchronous programming vs concurrent programming vs parallel programming.

- **Parallel programming** is about doing multiple things at the same time. In order to do multiple things at the same time, a computer needs to be able to process multiple streams on instructions simultaneously. Most computers these days have multi-core processors with 2 or 4 cores that enables parallel processing of code.
- **Concurrent programming** is about using abstractions within a program that may enable things to happen in parallel. If things actually happen in parallel depends on the runtime operating system and underlying hardware. Virtual parallel programming
- **Asynchronous programming** is simply doing something else while you are waiting for some another task to finish

Callbacks and higher-order functions

- In order to do asynchronous operations, we need to use callbacks
- A callback is simply a function or a method that we pass to another function, and we expect the callback to be called at some later time
- A function that takes another function as an argument, or returns a function as a result, is called a higher-order function
- In order to pass functions around, they need to be what's called first-class

First class functions

- A function is said to be first-class if you can use a function the same way as any other Python value

1. Assign a function or a method to a variable or object property
2. Pass functions as arguments to other functions or methods
3. Return functions as the result from other functions
4. Create functions dynamically at runtime

```
# Function returning a function
def addn(n):
   def addx(x):
       return n+x
   return addx
```

```
# Assign function to a variable
multiply = lambda a, b: a * b
```

```
# Function that takes a function as an argument
def square(x, multiply):
    return multiply(x, x)
```

```
# Create function dynamically at runtime
add_10 = eval("lambda x: x+10")
```

Continuation passing style (CPS)

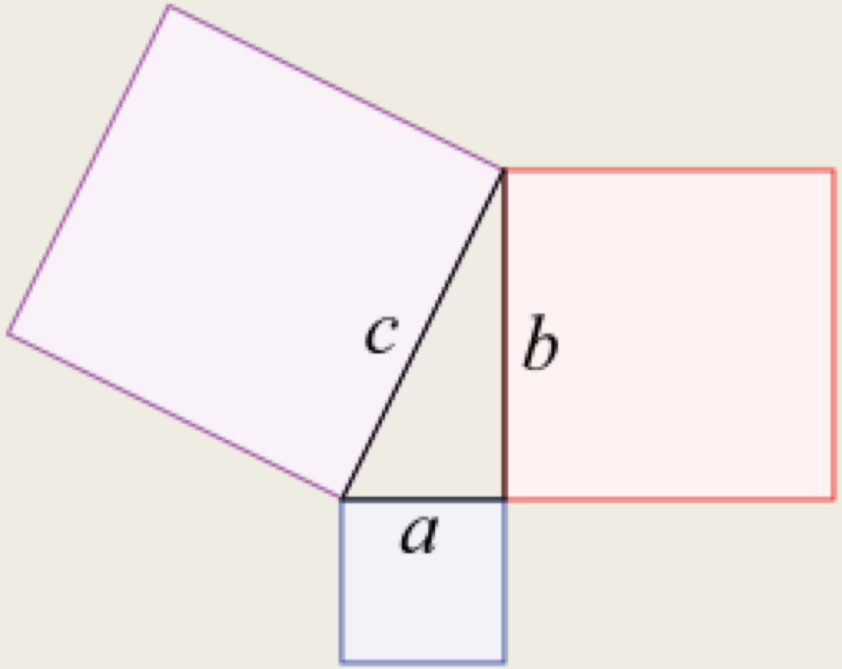
- Related to programming with callbacks is programming with Continuation Passing Style (CPS).
- This is a functional programming style where you don't return any values from your functions
- Instead you pass a continuation function that will be applied to the result

```
# Without continuation
def add_10(n):
    return n + 10
```

```
# With continuation
def add_10(n, cont):
    cont(n+10)
```

Show me the code …

- Pythagoras



Two problems ...

- You wanted to solve a problem with continuations. Now you have two problems!
- The code is unreadable.
- You might be able to write code that looks like this, but you will never be able to read it two weeks later.

- Can we do better?

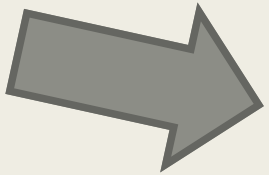
Thunks and trampolines

- A trampoline is a loop that invokes thunk-returning functions
- A thunk is basically just a function, with additional work that needs to be done
- We can think of the thunk as a suspended computation that needs to be invoked later

```
def square(x):
    return x*x

result = square(42) # A computation

thunk = lambda: square(42) # A suspended computation
thunk() # Invoke the computation
```

Factorial with and without trampoline

```
def factorial(n):
    if not n:
        return 1
    return n * factorial(n-1)
```



```
def trampoline(fn):
    while callable(fn):
        fn = fn()
    return fn
```

```
def factorial_cont(n, acc=1):
    if not n:
        return acc
    return lambda: factorial_cont(n - 1, n * acc)
```

Event-Based Programming

- Event-based programming is programming where the control of the program is not driven by the program itself, but by events triggered by the environment
- The environment is everything external to the program itself

```
# Self driven program
while True:
    data = get_data()
    process(data)
```

```
# Event driven program
while True:
   # Block until something happen
   event = wait_for_event()
   process(event)
```

What is an event?

- An event is something that has happened, that the program needs to handle in some way. Timers that have triggered
  - *Mouse moves and clicks*
  - *Keyboard input*
  - *Web requests, and other network events*
  - *Push notifications*
  - *Completion of an asynchronous operation*
  - *Sensor data such as temperature, fan speed, GPS data, accelerometer etc.*
- An ordered series of such events is called an event stream.

Event-loops and schedulers

- Fundamental to event-based programming is the concept of an event-loop
- An event-loop uses cooperative scheduling to process one event at a time
- This is very similar to our trampoline in that it unrolls the stack between invocations
- But instead of returning thunks, it instead uses continuations to schedule work that needs to be done later

```
def factorial(n, acc=1, schedule=None)
    if not n:
        return acc

    # Schedule a suspended computation
    schedule(lambda: factorial(n - 1, n * acc, schedule))
```

```
def scheduler(func, arg):
    ready = []

    def schedule(task):
        ready.append(task)

    result = func(arg, schedule=schedule)

    while ready:
        task = ready.pop(0)
        result = task()

    return result
```

Inline callbacks

- Writing callbacks easily becomes a mess when programming in Python.
- To chain multiple operations, we either have to use nested lambda functions, or define new functions or methods for each operation
- The chaining and control flow of these functions will be very hard to write, and is almost impossible to read
- What we like to do is to “inline” the callback below the asynchronous request, so the code looks similar to normal synchronous code

```
def long_running(cont):
    result = get_data()
    cont(result)

def callback(result):
    print(result)

long_running(callback)
```

```
result = long_running()
print(result)
```

Enhanced generators to the rescue

- We know that we can return values from any point within the generator function
- We can also send values into the generator and restart it at the last point where it yielded

- This is very interesting
- Could it be possible to yield long running operations, and then restart the function whenever the result is ready?

Are inline callbacks the solution?

- Inline callbacks combine callbacks, continuations, generators, trampolines and event-loops. Almost everything we have learned in this lecture
- It’s actually starting to get sort of nice
- We have hidden all the complexity of callbacks and continuations from the programmer
- The only problem is that things still look like a hack
- Sorts of becomes a new language within the language
- We are using generators for something they perhaps were not intended for

“… You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.” — Joe Armstrong

Asyncio

- Asyncio started out as the Tulip project and became part of Python standard library with Python-3.4
- Asyncio is defined in PEP- 3156
- Asyncio is also very inspired by modern projects such as Twisted and Tornado by using futures and coroutines
- The great news about asyncio is that Python now finally have an asynchronous programming model and asyncio is the platform

Futures

- A future is an abstraction that represent a future value
- Pull values vs push values

Pull	Push
n = 42	n = Future()

```
def get_42():
     return 42

n = get_42()
```

```
n = Future()

n.set_result(42)
```

Coroutines

- Coroutines are functions that can be entered, exited, and resumed at many different points
- This sounds a lot like our inline callbacks
- In fact coroutines in Python are simply just decorated enhanced generators

```
import asyncio

@asyncio.coroutine
def long_running(duration):
    yield from asyncio.sleep(duration)

@asyncio.coroutine
def main():
    yield from long_running(3)
    yield from long_running(3)
    yield from long_running(3)

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()
```

Async and await

- As we mentioned earlier, using yield within a coroutine sort of looks like a hack
- It feels like having a language within the language
- So Python 3.5 and PEP 0492 introduced 2 new keywords for dealing with asynchronous functions. They are:
  - ***Async**, mark a function or method as asynchronous.*
  - ***Await**, suspends execution until future is resolved, or coroutine completes*
- Async and await is not a feature specific to the Python programming language. It was first introduced first with C# 5.0 in 2011, and is now also used in other languages such Dart, Scala, TypeScript and is also proposed for JavaScript (ES7)

Asynchronous code can now be written the same way as synchronous code

- The great things about async and await is that you can program asynchronous programs exactly the same way as you used to program blocking synchronous programs.

Sync	Async
n = f(x)	n = await f_async(x)

```
import time

def long_running():
    time.sleep(1.0)

def main():
long_running()
```

```
import asyncio

async def long_running():
    await asyncio.sleep(1.0)

async def main():
    await long_running()

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()
```

Here be dragons

- What happens if some other part of the program modifies your object (shared state) while you are awaiting?
- https://github.com/python/cpython/blob/master/Lib/asyncio/locks.py

```
n = 42

async def work():
    print(n)
    await asyncio.sleep(10)
    print(n)
```

Summary

- **Async and await** is the combination of everything we have talked about:
  - *callbacks, continuations, iterables, iterators, generators, trampolines, event-loops, inline callbacks, coroutines*
- There's a lot we can learn from functional programming to improve imperative programming
- After all, async and await is really just syntactic sugar, right?

- Next time we will learn about Reactive programming