15-494/694: Cognitive Robotics

Dave Touretzky

Lecture 2:

VEX AIM Software Architecture

and

Python Control Structure

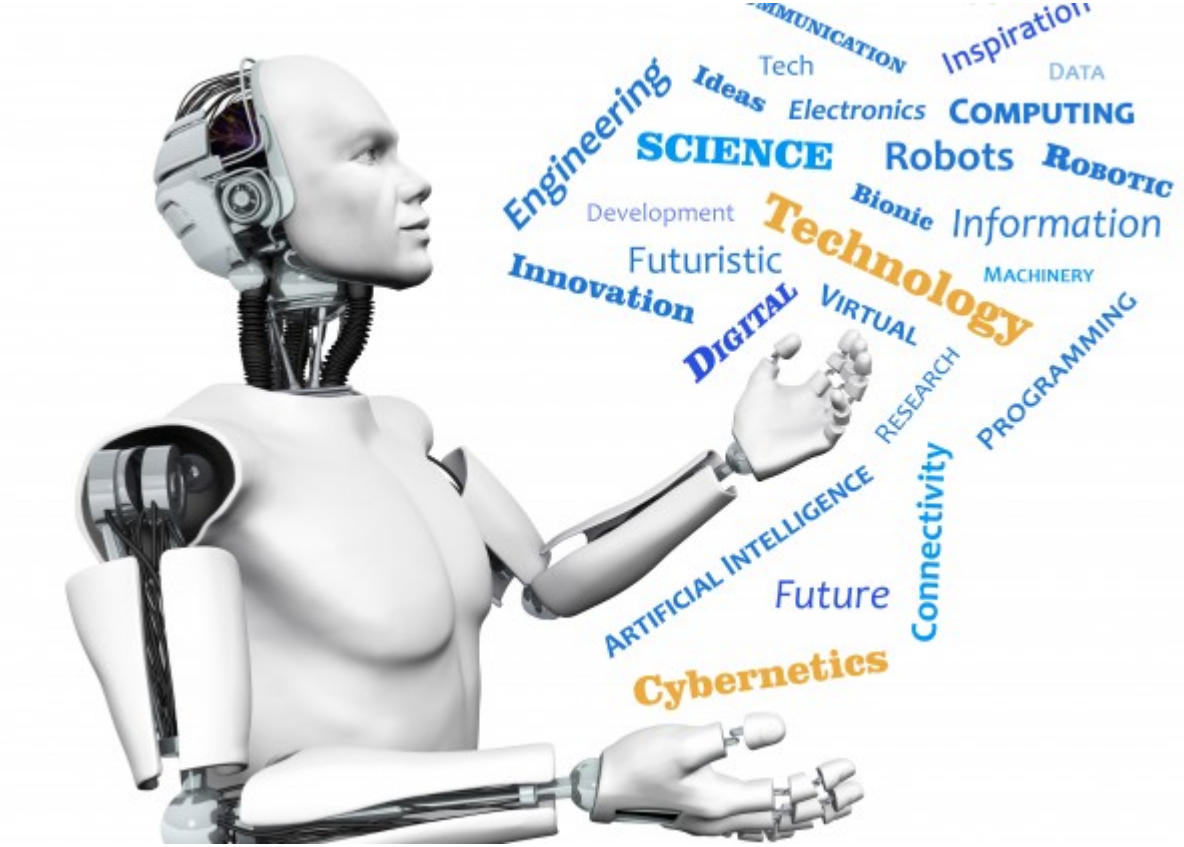


Image from http://www.futuristgerd.com/2015/09/10

Robot Software Architecture

- A robot is a complex collection of interacting hardware/software systems.
- Example: navigation isn't just motion.
  - Need vision to find landmarks.
  - Head + body motion to point the camera.
- Layers of control:
  - Low level: control one actuator
  - Middle level: coordinate multiple actuators (e.g., wheels and kicker) for one task.
  - High level: goal-directed behaviors.

Python Control Concepts

- To understand the vex-aim-tools architecture, you must be familiar with:
  - Iterators
  - Generators
  - Coroutines
  - Asyncio: event loops and tasks
  - Threads

Iterators

```
>>> nums = [1,2,3,4]
>>> for x in nums: print(f'x = {x}')
x=1
x=2
x=3
x=4
>>> [x*x for x in nums]
[1, 4, 9, 16]
```

list comprehension

What Makes an Object Iterable?

Defines an __iter__() method that returns an iterator.

```
>>> nums.__iter__
<method-wrapper '__iter__' of list object at 0x7ffa366baf48>

>>> nums.__iter__()
<list_iterator object at 0x7ffa34aa3c88>
```

What Is an Iterator?

References a sequence and defines a __next__() method that returns the next item or raises StopIteration if there are no more items.

```
>>> a = nums.__iter__()
>>> a.__next__()
1
>>> a.__next__()
2
```

StopIteration

```
>>> a.__next__()
3

>>> a.__next__()
4

>>> a.__next__()
Traceback: … StopIteration
```

How a For Loop Works

```
for x in nums: print(f'x = {x}')
```

```
it = nums.__iter__()
try:
  while True:
    x = it.__next__()
    print(f'x = {x}')
except StopIteration:
  pass
```

Lots of Things Are Iterable

```
>>> '__iter__' in dir([1,2,3])
True
```
list

```
>>> '__iter__' in dir(range(3,5))
True
```
range

```
>>> '__iter__' in dir({1,2,3})
True
```
set

```
>>> '__iter__' in dir({'foo' : 3})
True
```
dictionary

Make Your Own Iterable Thing

Needs an __iter__ method.

```
class MyIterable():

  def __init__(self,vals):
    self.vals = vals


  def __iter__(self):
    return MyIterator(self.vals)
```

Make Your Own Iterator

Needs a __next__ method.

```
class MyIterator():
  def __init__(self,vals):
    self.vals = vals
    self.index = 0

  def __next__(self):
    if self.index == len(self.vals):
      raise StopIteration
    else:
      self.index += 1
      return self.vals[self.index-1]
```

Testing MyIterable

```
>>> a = MyIterable([1, 2, 3, 4])
>>> for x in a: print(f'x = {x}')
x = 1
x = 2
x = 3
x = 4

>>> [x**3 for x in a]
[1, 8, 27, 64]
```

Generators

- Generators are *coroutines* that suspend their state using the **yield** keyword.
- Generators are represented by **generator** objects instead of functions.
- Generators can be used either as *producers* (similar to iterators) or as *consumers*.

Generator As Producer

```
def myproducer(vals):
  print('myproducer called')
  index = 0
  while index < len(vals):
    print('yielding')
    yield vals[index]
    index += 1
  raise StopIteration
```

Because “yield” appears in myproducer, calling myproducer doesn't actually run the function; it returns a generator object.

Generator As Producer

```
>>> g = myproducer(['foo','bar'])
<generator object myproducer at …>


>>> next(g)
myproducer called
yielding
'foo'


>>> next(g)
yielding
'bar'
```

Generator Expressions

Like a list comprehension, but uses parentheses instead of brackets: lazy.

```
>>> g = (x**2 for x in [1,2,3,4,5])
<generator object <genexpr> at …>

>>> next(g)
1

>>> g.__next__()
4
```

list() exhausts a generator

```
>>> g
<generator object <genexpr> at …>

>>> list(g)
[9, 16, 25]
```

Generator As Consumer

```
def myconsumer():
  print('myconsumer called')
  try:
    while True:
      x = yield
      print(f'{x} squared is {x**2}')
  except GeneratorExit:
    print('Generator closed.')
```

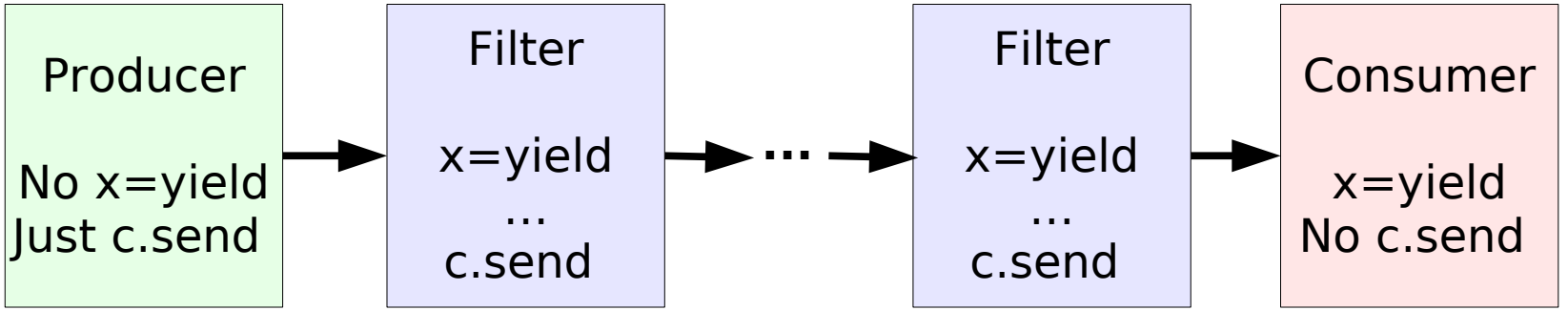
A statement 'x = yield' marks a *consumer* generator, which must be primed.

Generator As Consumer

```
>>> c = myconsumer()
<generator object myconsumer at …>
>>> c.send(None)
myconsumer called
>>> for x in range(1,5): c.send(x)
1 squared is 1
2 squared is 4
…
>>> c.close()
Generator closed.
```

Generator Pipeline

Generators can be chained together for complex processing tasks.



That's all we're going to say about generators. What about coroutines?

Coroutines Since Python 3.5

- In computer science, coroutines are procedures that repeatedly cede control to other coroutines and get it back again.

- In CS terms, Python generators can be regarded as coroutines: they “yield”.

- But in Python, coroutines are part of the **asyncio** module and do *not* use yield. So generators are *not* “coroutines” in Python.

asyncio

- A Python coroutine is defined using the keywords **async def** instead of the usual **def**.
- Coroutines use the **await** keyword to cede control:

  await mycor()

  and to receive values:

  x = **await** mycor()
- They return a value using **return**.
- Coroutines cannot use **yield**.

asyncio event loop

- The **asyncio** module provides for the *asynchronous* execution of coroutines. How?
- asyncio provides a task scheduler called an *event loop*.
- You can create tasks manually with loop.create_task().
- You can also give the loop a coroutine object and it will create a task for you.

Coroutine Example

```
import asyncio


async def yourcor(i):
  await asyncio.sleep(1)
  return i**2


async def mycor():
  for i in range(1,5):
    print(f'i={i}', end='')
    x = await yourcor(i)
    print(f' x={x}')
```

Testing the Coroutine Example

```
>>> loop = asyncio.get_event_loop()
<_UnixSelectorEventLoop …>

>>> c = mycor()
<coroutine object mycor at …>

>>> loop.run_until_complete(c)
i=1 x=1
i=2 x=4
i=3 x=9
i=4 x=16
```

Adding Tasks To the Queue

```
>>> t = loop.create_task(yourcor(5))
<Task pending coro=yourcor() …>


>>> loop.run_until_complete(t)
25
```

Scheduling Non-Coroutines

- What if you want the event loop to execute a regular function instead of a coroutine?
- Use loop.call_soon() or loop.call_later()
- Instead of creating a Task, this creates a Handle or TimerHandle and schedules it for immediate or delayed execution.

Scheduling Non-Coroutines

```
def goof(i):
  print('i=', i)


>>> loop.call_soon(goof, 150)
<Handle goof(150) at …>
>>> loop.call_later(3,goof,250)
<TimerHandle when=…>


>>> loop.run_forever()
i=150
i=250
```

Futures

- A Future is an object representing a value that might not have been computed yet.
- Created by loop.create_future()
- A coroutine can return a Future and then later some other coroutine can fill in the value.
- You can test whether a Future has completed, or set up a callback that will be called when the Future completes.

Threads

- Threads are lightweight units of execution within a process that run simultaneously.
- Threads share one address space.
- If two threads modify the same memory at the same time, bad things may happen.
- “Thread-safe” code uses interlocks to prevent this.
- loop.call_soon_threadsafe() lets secondary threads access the event loop.

VEX AIM SDK Robot Interaction

- Uses websockets to talk to the robot.
- Four secondary threads:
  - **Image thread** receives camera frames
  - **Status thread** receives status updates:
    - Sensor values and odometry
    - Object detection results (“aivision”)
    - Actions in progress (motion, sound)
  - **Command thread** transmits commands to the robot
  - **Audio thread** transmits audio files to the robot

VEX AIM SDK: Files

- aim.py contains most of the SDK
- vex.py contains type definitions and important constants

VEX AIM SDK: Low-Level Control

- The SDK only provides simple, low-level primitives.
- Example: how to drive forward 50 mm?
- robot.move_for(50, 0)
  - Waits until move is complete.
  - No other processing can take place while waiting. *This is not a good thing!*
- robot.move_for(50, 0, wait=False)
  - Doesn’t wait. Now it’s your responsibility to notice when the robot stops moving.

vex-aim-tools

- Built on top of the VEX AIM SDK.
- Supports event-based programming so actions run asynchronously and you can be notified when the robot finishes an action.
- Runs an asyncio event loop in a secondary thread so the main thread is available for the Python REPL: debug your program while it runs.
- User programs run in the secondary thread, using the event loop.

vex-aim-tools

- Provides asynchronous services:
  - speech recognition
  - speech generation
  - GPT-4 interface
  - other computationally intensive things
- Runs visualization tools (camera viewer, worldmap viewer, etc.) in their own threads.

Does This Look Like Fun? No???

- Explicitly managing asynchronous actions and events could be a real pain.
- Is there a better way?

- State machines. See next lecture.