15-494/694: Cognitive Robotics

Dave Touretzky

Lecture 2:

Cozmo Software Architecture

and

Python Control Structure

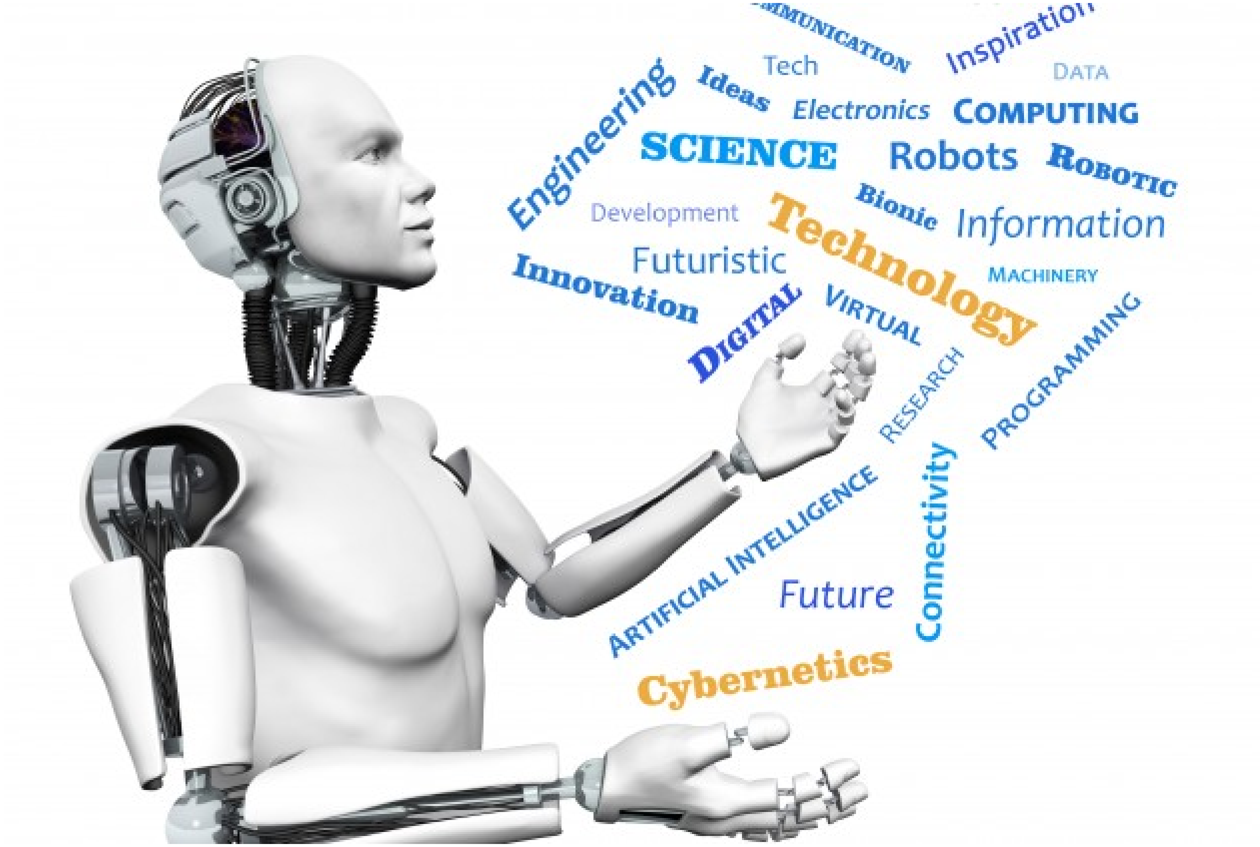


Image from http://www.futuristgerd.com/2015/09/10

Cozmo 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., head and wheels) for one task.
  - High level: goal-directed behaviors.

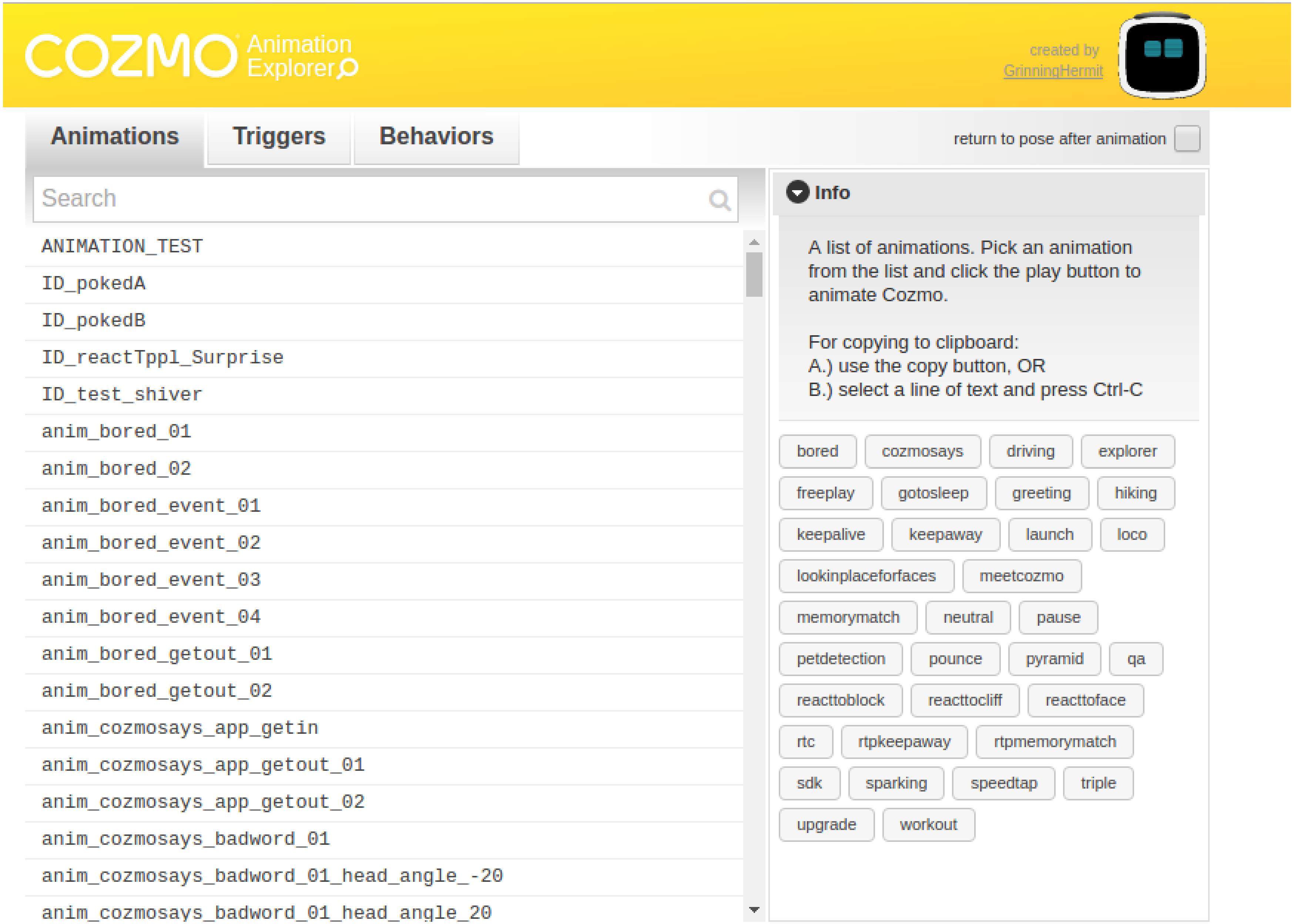
Control Levels in Cozmo (1)

- **Actions**: basic operations that focus on one effector but can optionally include some gratuitous animations.
  - drive_forward
  - turn_in_place
  - set_head_angle
  - move_lift
  - say_text

Control Levels in Cozmo (2)

- **Animations**: short behavior sequences that involve a combination of body motions, facial expressions, and sound effects.
- Designed by former Pixar animators.
- In SDK version 1.2.1 there are 955 animations, organized into groups.
- See `robot.conn.anim_names` for the list.
- Use the Cozmo Animation Explorer tool to try them out.

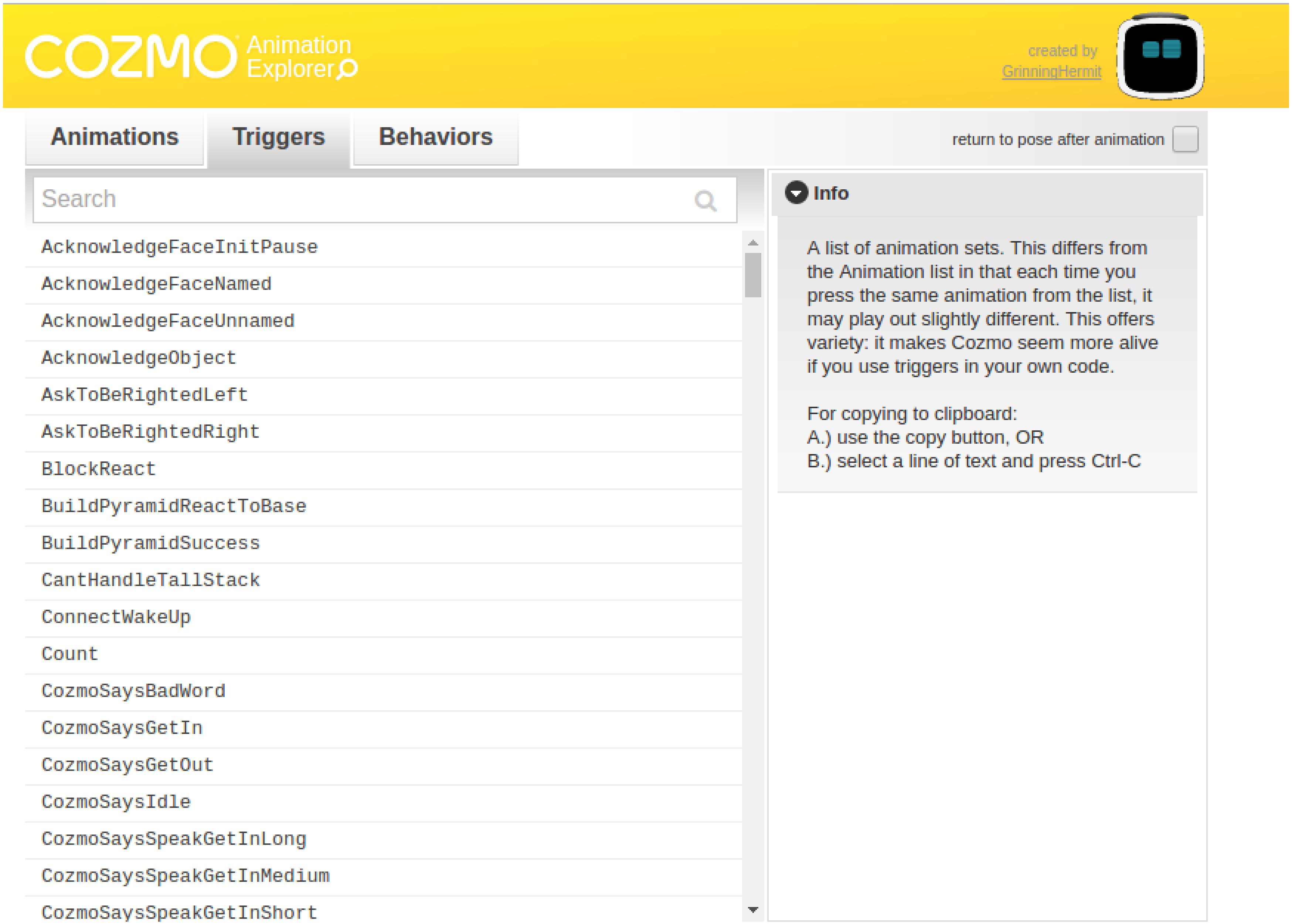
955 Animations



Control Levels in Cozmo (2.5)

- **Animation Triggers**: Families of animations that are variants on a theme.
- Playing a trigger will select one animation at random from the family.
- In version 1.2.1 of the SDK there are 544 triggers.
- `dir(cozmo.anim.Triggers)`
- Both animations and triggers have well-defined completion points.

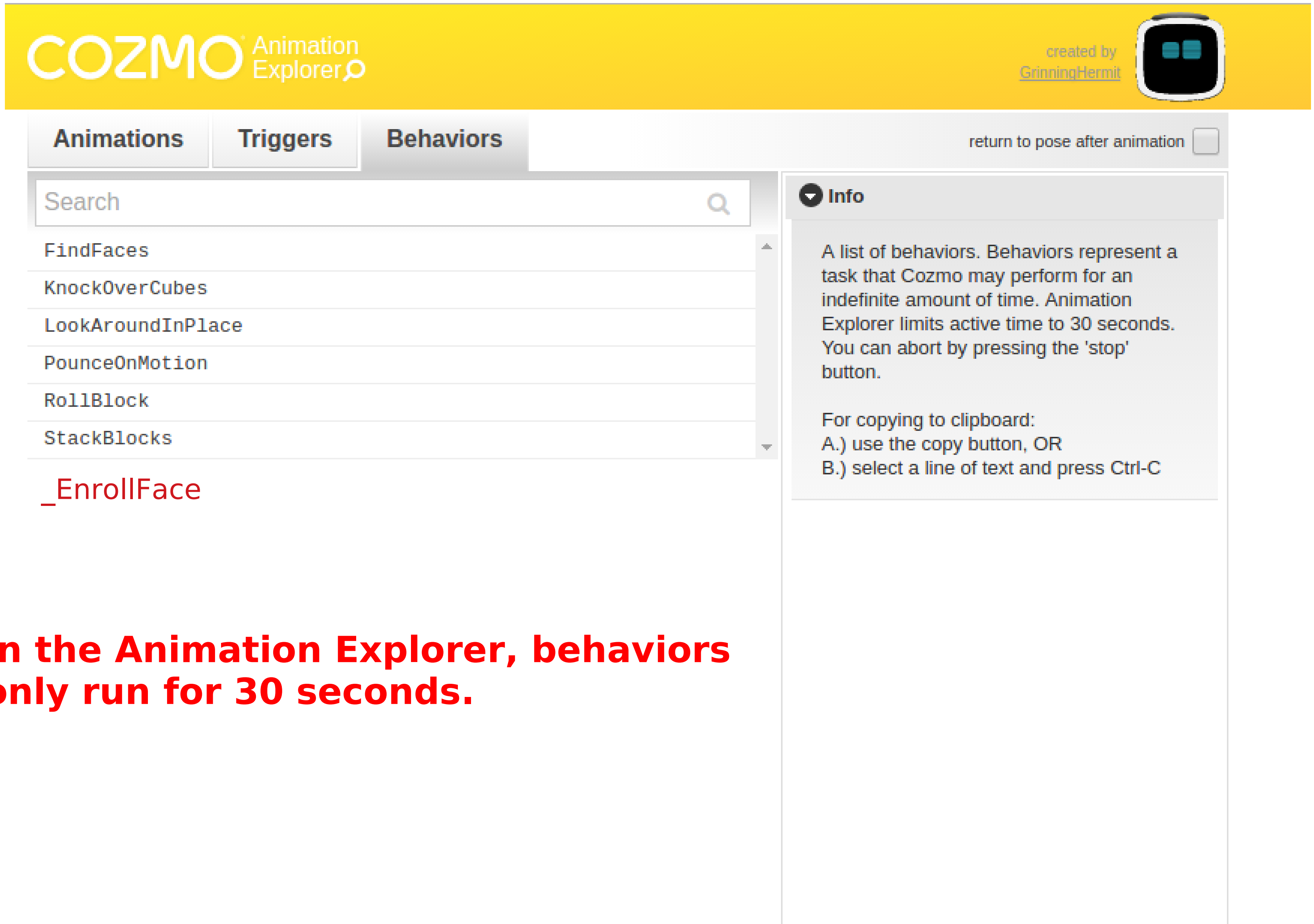
544 Animation Triggers



Control Levels in Cozmo (3)

- Behaviors: Complex operations that try to accomplish a goal.
- Only seven defined so far:
  - Vision: FindFaces, LookAroundInPlace, _EnrollFace
  - Manipulation: KnockOverCubes, RollBlock, StackBlocks
  - Human interaction: PounceOnMotion
- Behaviors use multiple animations.
- Behaviors never complete; they must be explicitly stopped.

Only 7 Behaviors So Far



_EnrollFace

In the Animation Explorer, behaviors only run for 30 seconds.

Python Control Concepts

- The Cozmo SDK is written in industrial strength Python 3.5.
- To understand the SDK, you must be familiar with:
  - Iterators
  - Generators
  - Coroutines
  - Asyncio: futures, tasks, handles, event loops

Iterators

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

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('x=%s' % x)
```

```
it = nums.__iter__()
try:
  while True:
    x = it.__next__()
    print('x=%s' % x)
except StopIteration:
  pass
```

Lots of Things Are Iterable

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

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

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

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

Make Your Own Iterable Thing

```
class MyIterable():

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


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

Make Your Own Iterator

```
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('x=%s' % 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('%s squared is %s' %
            (x, x**2))
  except GeneratorExit:
    print('Generator closed.')
```

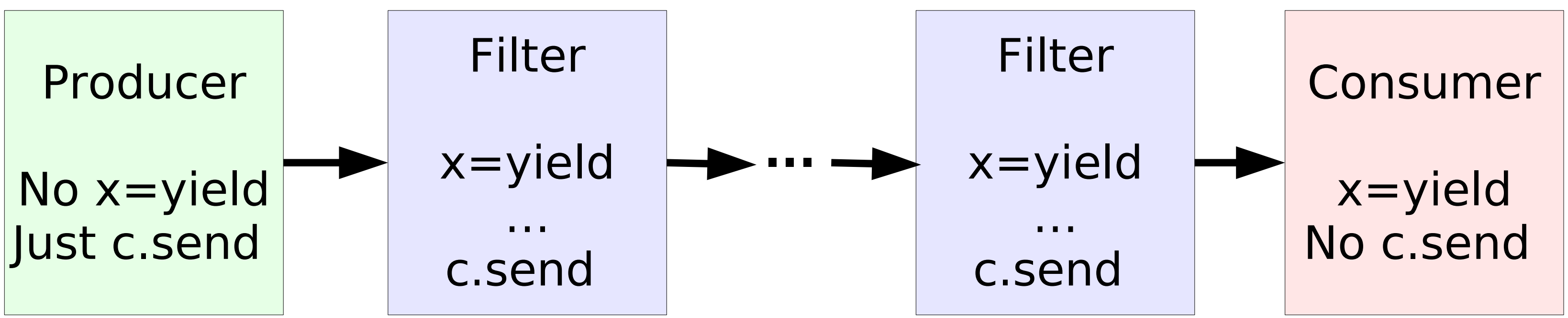
A statement 'x=yield' marks a *consumer* generator, which must be primed.

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?

Python Will Drive You Crazy

- Python changes every year.
- This has been going on for a long time.
- The terminology changes as well.
- Result: Python is confusing as hell.
- Reading tutorials written several years ago will drive you crazy.
- Coroutines are a prime example.

Newbie: *“How do coroutines work?”*

Expert: *“Well, in Python 2.7 it did this, but then in Python 3.3 it did that, and now in Python 3.5 it does this other thing, but in Python 3.7 it's going to ...”*

Newbie: “Kill me now.”

History of Python Coroutines

- You don't want to know.

- Stuff to forget about:

  @coroutine decorator

  @asyncio.coroutine decorator

  “generators are coroutines” – no longer

Coroutines in Python 3.5

- In computer science, coroutines are procedures that repeatedly cede control to their caller and get it back again.
- In CS terms, Python generators are coroutines. They use “yield”.
- In Python 3.5 and up, “coroutine” has a more specific meaning, and generators are *not* coroutines.

Coroutines in Python 3.5

- The asyncio module provides a kind of scheduler called an *event loop*.
- Coroutines are asynchronously executing procedures, ceding control to each other or the event loop that manages them.
- Coroutines in Python 3.5 are defined with **async def** instead of the usual **def**.
- They use the **await** keyword to cede control until the thing they're awaiting has finished. They cannot use **yield**.

Coroutine Example

```
import asyncio

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

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

Testing the Coroutine Example

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

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

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

Tasks and Futures

- A **Future** is an object representing an asynchronous computation that may not yet have completed.

- You can attach handlers to futures that will be notified when the future completes.

- A **Task** is a kind of Future that is managed by an event loop.

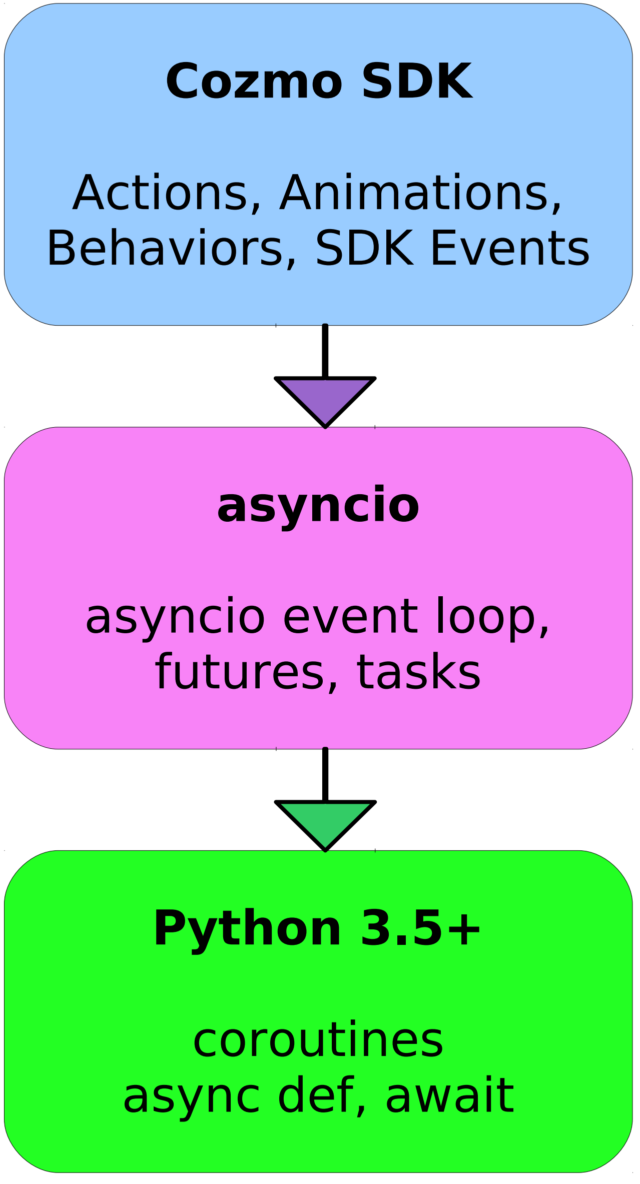
Adding Tasks To the Queue

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


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

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
```

The Big Picture



SDK and the Event Loop

- The Cozmo SDK includes an asyncio event loop which is accessible at `robot.loop`.
- The Cozmo SDK provides its own classes for representing actions, animations, etc. as tasks managed by this event loop.
- The SDK (not asyncio) `wait_for_completed()` method waits until the event loop has completed the task.

Cozmo Actions Are Tasks

```
#!/usr/bin/python3

import asyncio
import cozmo

async def mytalker(robot):
  action = robot.say_text('hello')
  print('act =', action)
  coro = action.wait_for_completed()
  print('coro =', coro)


cozmo.run_program(mytalker)
```

Cozmo Actions Are Tasks

```
$ ./mytalker.py
… [set up connection to robot … ]

act = <SayText state=action_running …>
coro = <coroutine object
         Action.wait_for_completed …>
```

The SDK's Event Dispatcher

- The SDK defines a collection of robot events (e.g., an object has become visible, or a cube is tapped).
- The SDK includes its own event dispatcher, and a way to set up listeners for SDK events.
- Don't confuse this with the asyncio event loop. Despite the name “event loop”, asyncio doesn't have events. The SDK does.

Threads

- The Cozmo Python SDK is single-threaded.
- The REPL runs in a separate thread.
- But cozmo-tools uses multiple threads for visualization tools such as the world map viewer.
- Not thread-safe, but close enough.

Does This Look Like Fun? No???

- Explcitly managing coroutines, tasks, etc. looks like it could be a real pain.
- Is there a better way?

- State machines. See next lecture.