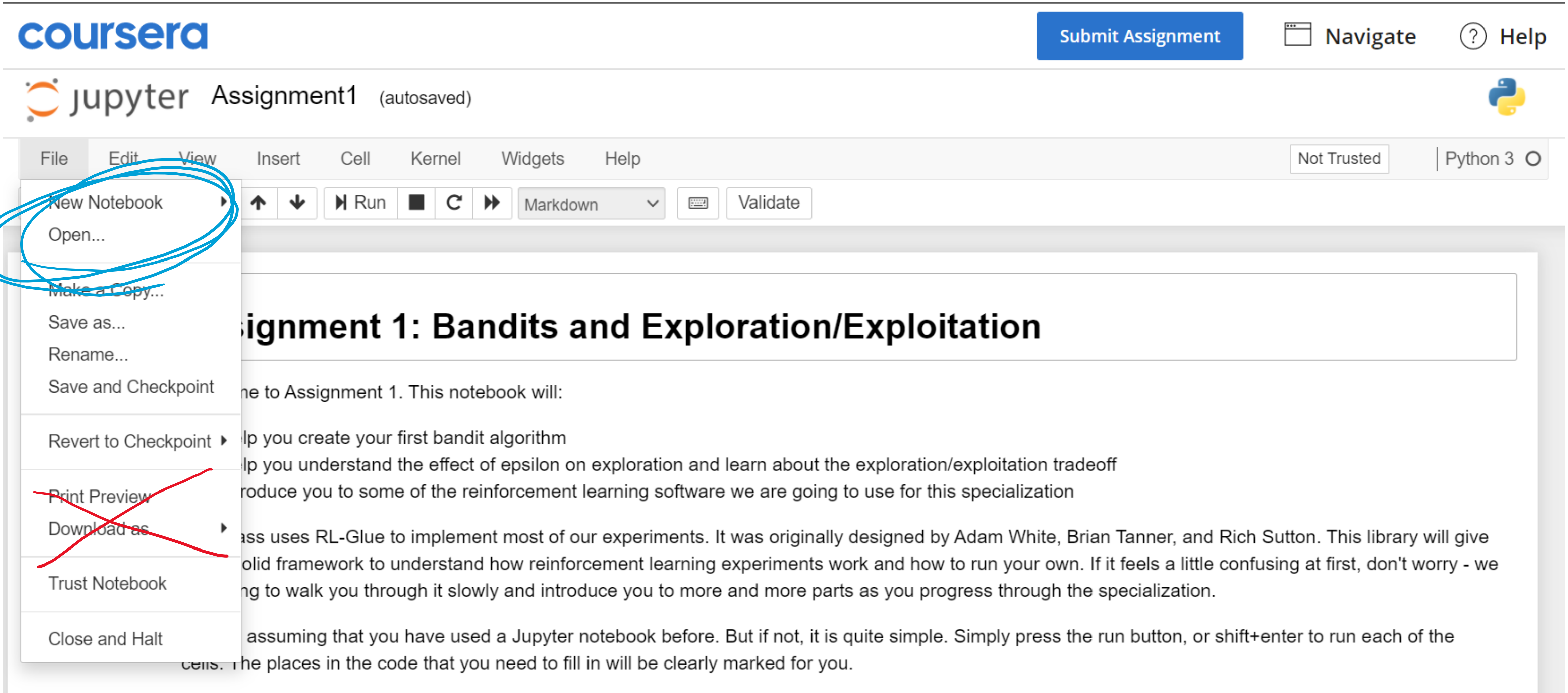
CMPUT 365 – Python3 Review

Preliminaries

A few important topics we *won’t* talk about today:

* pip install …
* local jupyter
* python modules (mostly)
* code organization (mostly)
* programming style (mostly)

Downloading code from Coursera

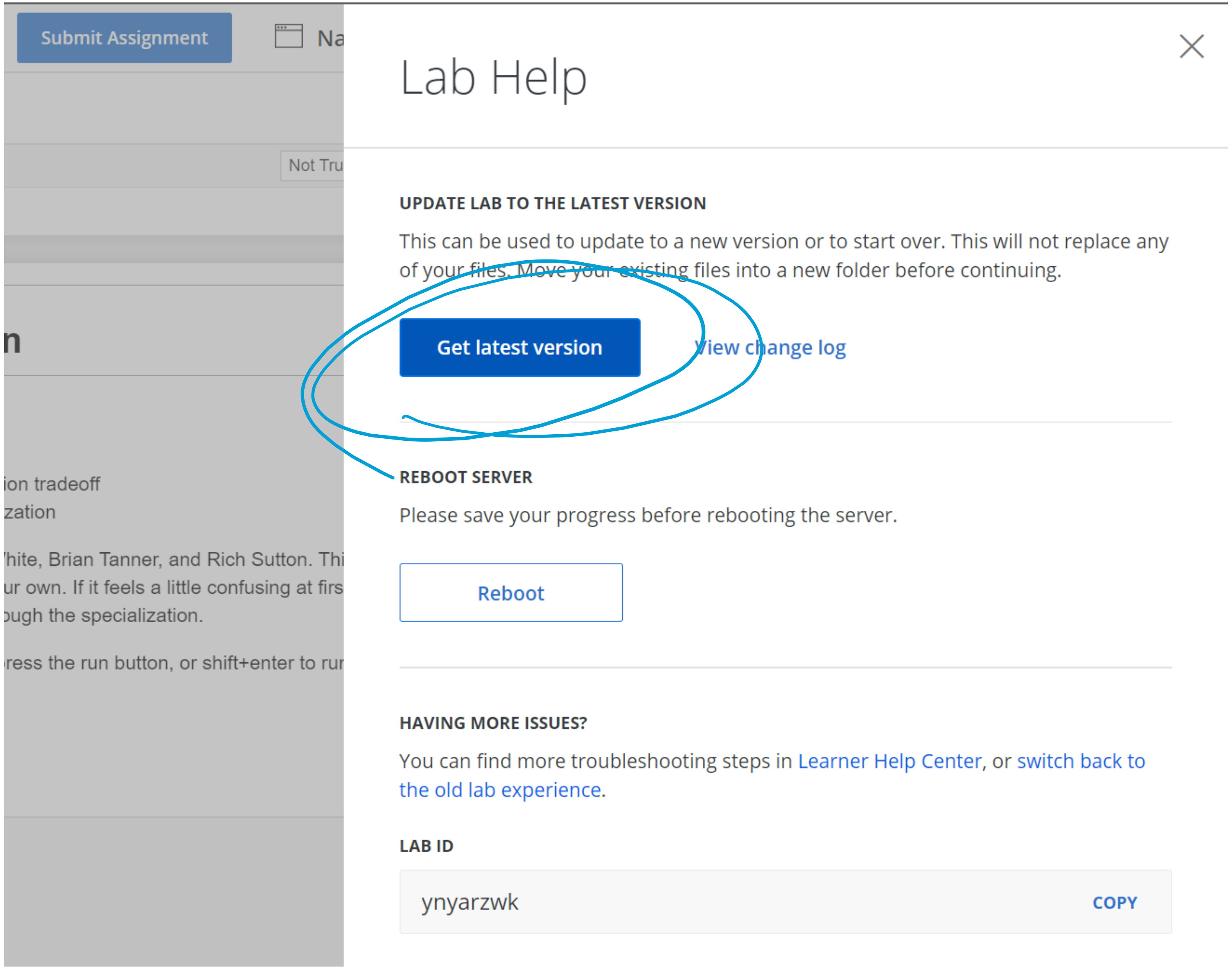


Downloading code from Coursera

Burn it to the ground

Burn it to the ground

Burn it to the ground



Preliminaries

1) Make it work
2) Make it pretty

3) Make it fast

A first example

```python
import numpy as np
from RlGlue.environment import BaseEnvironment

class Environment(BaseEnvironment):
    def __init__(self, arms):
        super().__init__()

        self.arms = arms
        self.mean_rewards = np.random.normal(loc=0, scale=1, size=self.arms)

    def step(self, a):
        mean = self.mean_rewards[a]
        return np.random.normal(mean, 1)
```

A first example

Imports

```python
import numpy as np
from RlGlue.environment import BaseEnvironment
```

Classes + inheritance

Constructor

```python
class Environment(BaseEnvironment):
    def __init__(self, arms):
        super().__init__()
```

Instance attributes

```python
        self.arms = arms
        self.mean_rewards = np.random.normal(loc=0, scale=1, size=self.arms)
```

Instance methods

```python
    def step(self, a):
        mean = self.mean_rewards[a]
        return np.random.normal(mean, 1)
```

Variables

```
a = 'hi'
b = 22
c = [1, 2, 3]
d = { 'cow': 'black', 'pig': 'pink' }
e = d
```

Variables

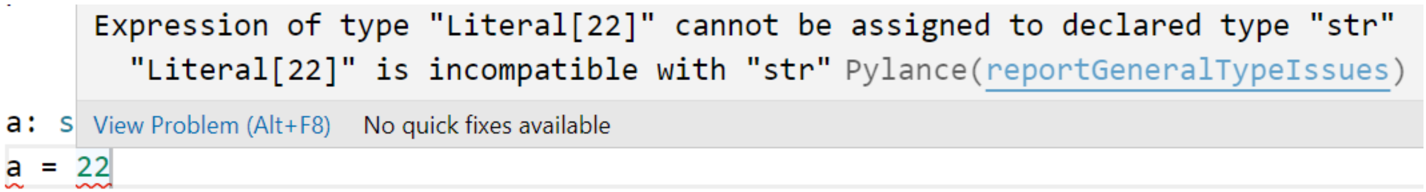
```
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>>
```

Strongly, Dynamically (usually) Typed

```python
a = 'hi'
a = 22
type(a)    # => <class 'int'>
type(22)   # => <class 'int'>
type('hi') # => <class 'str'>
```

Strongly, Statically (but not really) Typed

```python
a: str = 'hi'
a = 22
```



Functions

Note indentation

Only one return value

```
def f(x):
    return x + 1

def g(x, y):
    return x + y

def h(x, y):
    return (x, y)
```

Aside: many tutorials are wrong on this point (for example the one whose organizational structure inspired this slideshow)

Functions

```python
def h(x, y):
    return x, y

out = h(1, 2)
print(out) # => (1, 2)
type(out)  # => <class 'tuple'>

a, b = h(1, 2)
```

Still returns a Tuple type, just uses implicit tuple-destructuring to make it *look* like it returns two values

Aside: many tutorials are wrong on this point (for example the one whose organizational structure I borrowed this material)

Functions

```python
def f(x, y=1):
    return x + y

f(3) # => 4
f(3, 4) # => 7
```

Functions

```
def f(x, y=1):
    return x + y

f(x=1, y=3) # => 4
f(y=2, x=3) # => 5
f(1, y=6)   # => 7
```

But please don't

```
f(*[1,2])                # => 3
f(**{'x': 3, 'y': 8}) # => 11
```

Pass-by-value vs. Pass-by-reference

No return value →

```
from typing import List

def f(x: List[int]):
    x[0] = 1

a = [0, 1, 2, 3]
f(a)
print(a) # => [1, 1, 2, 3]
```

Pass-by-value vs. Pass-by-reference

```python
def f(x: float):
    x = 3.14

a = 2.71
f(a)
print(a) # => 2.71
```

No change

Pass-by-value vs. Pass-by-reference

```python
from typing import List

def f(x: List[str]):
    x = ['hi', 'there']

a = ['what', 'gonna', 'happen', '??']
f(a)



print(a) # => ['what', 'gonna', 'happen', '??']
```

Pass-by-value vs. Pass-by-reference

```python
a = 1
def f(x):
    a = 2

f(3)
print(a) # => 1
```

Pass-by-value vs. Pass-by-reference

To recap:

* Primatives are *always* pass-by-value
* Variables are *defined* as locally scoped
* Variables in the global scope can be *read*
* Reaching “inside” the data and changing a value will mutate the data

Dictionaries

```
d = {
    'hi': 'there',
    'data': 22,
    'inner': {
        'more': 'data',
    },
}
```

Dictionaries

```
d = {
    'hi': 'there',
    'data': 22,
    'inner': {
        'more': 'data',
    },
}
```

```
d['hi'] # => 'there'
d['inner'] # => { 'more': 'data' }
d['inner']['more'] # => 'data'

d['hi'] = -3
d['hi'] # => -3
```

Dictionaries

Tech specs:

* Dictionaries use a hash-table for the keys
* Any hashable value can be used as a key
* Reading is O(1)
* Changing values, adding keys, removing keys: O(1)

Classes and objects

Class name (and type)

Class instance (an object)

```python
class Agent:
    def __init__(self, arms, epsilon, stepsize):
        self.arms = arms
        self.means = np.zeros(arms)

        self.epsilon = epsilon
        self.stepsize = stepsize
```

Constructor function

Classes and objects

```python
class Agent:
    def __init__(self, arms, epsilon, stepsize):
        self.arms = arms
        self.means = np.zeros(arms)

        self.epsilon = epsilon
        self.stepsize = stepsize


a1 = Agent(10, 0.1, 0.01)
a2 = Agent(22, 0.5, -3)

a1.arms # => 10
a2.arms # => 22
```

Note only 3 arguments not 4. The `self` argument is supplied by the runtime on instantiation.

Classes and objects

```
class Agent:
    def __init__(self, arms, epsilon, stepsize):
        ...

    def update(self, r, a):
        self.means[a] = self.means[a] + self.stepsize * (r - self.means[a])

a = Agent(10, 0.1, 0.01)
a.update(1, 0)
```

Note only 2 arguments not 3. The `self` argument is supplied by the runtime again.

```
Agent.update(a, 1, 0)
```

But please don't

Aside: we’re all consenting adults here

```python
class HiddenStuff:
    def __init__(self):
        self._private_data = 0
        self.__super_private = 0
        self.public = 0
```

Lists and (np) arrays

```python
a = [1, 2, 3]
b = ['1', 2, {'thing': 3}]
c = list(range(100))

a.append(4) # => [1, 2, 3, 4] (in place)

d = a + [5] # => [1, 2, 3, 4, 5]
```

Lists and (np) arrays

```python
import numpy as np

a = np.array([1, 2, 3])
b = np.zeros(3)

c = a + b
d = a.dot(b)
```

Lists and (np) arrays

Tech specs:

* Lists are linked-lists: (data, pointer) -> (data, pointer) -> (data, pointer)
* Arrays are contiguous memory: pointer -> [data, data, data]
* Lists can hold arbitrary data of differing types
* (np) Arrays are limited to numerical data(ish) of the same type

* List lookup is O(n), but iteration is O(1) per-element
* List append is O(n)
* List deletion is O(1)

* Array lookup is O(1)
* Array append is O(n) (kinda)

For loops

Note indentation

```
for a in [1, 2, 3]:
    print(a) # => 1, then 2, then 3

print(a) # ew... but 3
```

For loops

```
for a in range(100):
    print(a) # => 0, then 1, then 2, then ..., then 99
```

List Comprehensions

```
a = []
for i in range(3):
    a.append(i)

print(a) # => [0, 1, 2]
```

List Comprehensions

```python
a = [i+3 for i in range(3)]

print(a) # => [3, 4, 5]
```

Notes:

* Technically faster; recall O(n) append
* Can contain pretty complex logic
* Always ask: should I?

List Comprehensions

```python
a = [i for i in range(3)]

print(a) # => [0, 1, 2]
```

1) Make it work
2) Make it pretty

3) Make it fast

Notes:

* Technically faster; recall O(n) append
* Can contain pretty complex logic
* Always ask: should I?

Vectors and matrices

```
a = np.zeros(3)
print(a) # => [0, 0, 0]

a = np.zeros((3, 3))
print(a) # =>
# [
#   [0, 0, 0],
#   [0, 0, 0],
#   [0, 0, 0],
# ]
```

Vectors and matrices

```
a = np.zeros((3, 3))

a[0, 0] = 1
print(a[0, 0]) # => 1

a[0, :] = [1, 2, 3]
print(a[0, 0]) # => 1
print(a[0, 1]) # => 2
print(a[0])    # => [1, 2, 3]
```

Broadcasting

```
a = np.zeros(3)

a = a + 1
print(a) # => [1, 1, 1]

a = a * 2
print(a) # => [2, 2, 2]
```

Linear algebra

```
X = np.ones((10, 5))

C = X.T.dot(X)
C = np.dot(X.T, X)

Cinv = np.linalg.inv(C)

P = X.dot(Cinv).dot(X.T)
```

Linear algebra

```
a = np.ones(3)      # => vector
b = np.zeros(3)     # => vector

A = np.ones((5, 3)) # => matrix

a.dot(b)   # => scalar
A.dot(a)   # => vector
A.dot(A.T) # => matrix
```

CAUTION: non-obvious roadblocks

```
a = np.ones((3, 1)) # => matrix that looks like col-vector
b = np.ones(3)      # => vector

a.dot(b)   # error! incompatible dimensions 1 and 3
b.dot(a)   # => vector that looks like scalar (vector-matrix product)
a.T.dot(b) # => vector that looks like scalar (matrix-vector product)

a = np.ones(3)
b = np.ones(3)

a.T.dot(b)     # => scalar
a.dot(b.T)     # => scalar! (not an outer product)
np.outer(a, b) # => matrix
```

Axes

```
A = np.array([
    [1, 2, 3],
    [2, 3, 4],
    [3, 4, 5],
    [4, 5, 6],
    [5, 6, 7],
])

print(A.shape) # => (5, 3)
np.mean(A, axis=0) # => [3, 4, 5]
np.mean(A, axis=1) # => [2, 3, 4, 5, 6]
```

Think “over rows”

Think “over cols”

Axes

```
A = np.zeros((runs, hypers, episodes, measures))

np.mean(A, axis=0) # => over "runs",     shape=(hypers, episodes, measures)
np.mean(A, axis=1) # => over "hypers",   shape=(runs, episodes, measures)
np.mean(A, axis=2) # => over "episodes", shape=(runs, hypers, measures)
```

Random numbers

```python
def f():
    np.random.seed(0)
    return np.random.randint(0, 10)

f() # => 2
f() # => 2
f() # => 2
...
f() # => 2
```

Random numbers

```
def f():
    return np.random.randint(0, 10)

np.random.seed(0)
f() # => 2
f() # => 8
f() # => 1
...
f() # => 3

np.random.seed(0)
f() # => 2
f() # => 8
f() # => 1
...
f() # => 3
```

Random numbers

```
np.random.normal(loc=0, scale=1)
np.random.normal(0, 1)
np.random.normal(0, 1, size=(3, 5))

x = np.random.rand() # => x \in [0, 1)
x = np.random.randint(0, 10) # => x \in [0, 10)
x = np.random.choice([2, 4, 6, 8])

a = [1, 2, 3, 4]
np.random.shuffle(a) # in place
a # => [3, 4, 2, 1]
```

Notes:

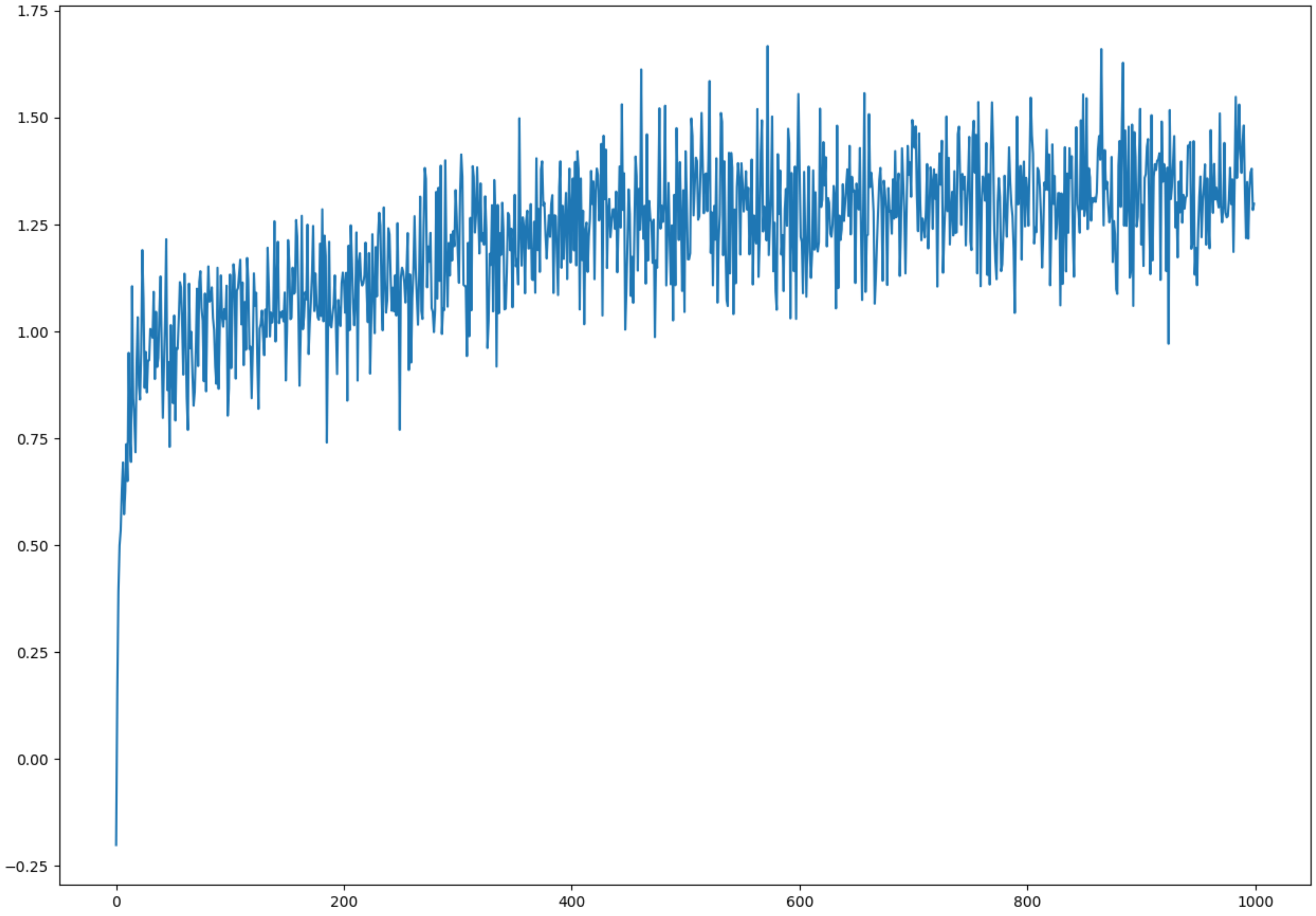
* *Every* call to the rng changes the internal rng state
* Do **not** use the built-in `random` library, *always* use np.random
* Avoid calling the rng when it is not necessary
* Every program in this course should be deterministic meaning you should **always** set the rng seed

Putting it together

```python
import numpy as np
from RlGlue.environment import BaseEnvironment

class Environment(BaseEnvironment):
    def __init__(self, arms):
        super().__init__()

        self.arms = arms
        self.mean_rewards = np.random.normal(0, 1, self.arms)

    def step(self, a):
        mean = self.mean_rewards[a]
        return np.random.normal(mean, 1)

class Agent:
    def __init__(self, arms, epsilon, stepsize):
        self.arms = arms
        self.means = np.zeros(arms)

        self.epsilon = epsilon
        self.stepsize = stepsize

    def act(self):
        estimated_best = np.argmax(self.means)

        if np.random.rand() < self.epsilon:
            return np.random.randint(0, self.arms)

        return estimated_best

    def update(self, r, a):
        self.means[a] = self.means[a] + self.stepsize * (r - self.means[a])
```

```python
rewards = []

for run in range(100):
    np.random.seed(run)

    env = Environment(10)
    agent = Agent(10, 0.1, 0.1)

    run_rewards = []
    for _ in range(1000):
        a = agent.act()
        r = env.step(a)

        agent.update(r, a)

        run_rewards.append(r)

    rewards.append(run_rewards)

import matplotlib.pyplot as plt
plt.plot(np.mean(rewards, axis=0))
plt.show()
```

Putting it together



Juptyer notebook gotchas

In [ ]:

```
# -----------
# Tested Cell
# -----------
# The contents of the cell will be tested by the autograder.
# If they do not pass here, they will not pass there.

test_array = [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
assert argmax(test_array) == 8, "Check your argmax implementation returns the index of the largest value"

# set random seed so results are deterministic
np.random.seed(0)
test_array = [1, 0, 0, 1]

counts = [0, 0, 0, 0]
for _ in range(100):
    a = argmax(test_array)
    counts[a] += 1

# make sure argmax does not always choose first entry
assert counts[0] != 100, "Make sure your argmax implementation randomly chooses among the largest values."

# make sure argmax does not always choose last entry
assert counts[3] != 100, "Make sure your argmax implementation randomly chooses among the largest values."

# make sure the random number generator is called exactly once whenver `argmax` is called
expected = [44, 0, 0, 56] # <-- notice not perfectly uniform due to randomness
assert counts == expected
```

Scope bleeding

Juptyer notebook gotchas

In [ ]:

```
# -----------
# Graded Cell
# -----------
class GreedyAgent(main_agent.Agent):
    def agent_step(self, reward, observation=None):
        """
        Takes one step for the agent. It takes in a reward and observation and
        returns the action the agent chooses at that time step.

        Arguments:
        reward -- float, the reward the agent recieved from the environment after taking the last action.
        observation -- float, the observed state the agent is in. Do not worry about this as you will not use it
                              until future lessons
        Returns:
        current_action -- int, the action chosen by the agent at the current time step.
        """
        ### Useful Class Variables ###
        # self.q_values : An array with what the agent believes each of the values of the arm are.
        # self.arm_count : An array with a count of the number of times each arm has been pulled.
        # self.last_action : The action that the agent took on the previous time step
        #######################

        # Update Q values Hint: Look at the algorithm in section 2.4 of the textbook.
        # increment the counter in self.arm_count for the action from the previous time step
        # update the step size using self.arm_count
        # update self.q_values for the action from the previous time step

        # YOUR CODE HERE
        raise NotImplementedError()

        # current action = ? # Use the argmax function you created above
        # YOUR CODE HERE
        self.q_values[a] = self.q_values[a] + self.stepsize * delta

        self.last_action = current_action

        return current_action
```

Scope bleeding

Juptyer notebook gotchas

[1]
```
a = 0
print(a)
```

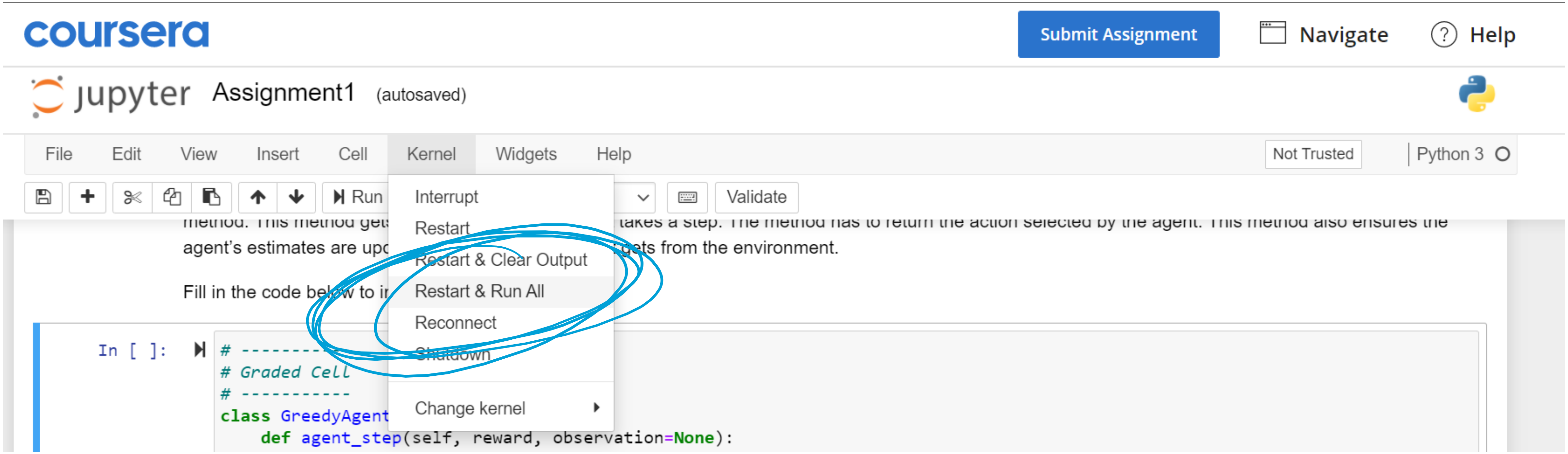
[4]
```
a += 1
```

[5]
```
print(a)
```

```
a = 0
print(a)

a += 1

print(a)
```

```
a = 0
print(a)

a += 1
a += 1
a += 1

print(a)
```

Juptyer notebook gotchas



Running the tests

In [ ]:

```
# --------------
# Debugging Cell
# --------------
# Feel free to make any changes to this cell to debug your code

# build a fake agent for testing and set some initial conditions
np.random.seed(1)
greedy_agent = GreedyAgent()
greedy_agent.q_values = [0, 0, 0.5, 0, 0]
greedy_agent.arm_count = [0, 1, 0, 0, 0]
greedy_agent.last_action = 1

action = greedy_agent.agent_step(reward=1)

# make sure the q_values were updated correctly
assert greedy_agent.q_values == [0, 0.5, 0.5, 0, 0]

# make sure the agent is using the argmax that breaks ties randomly
assert action == 2
```

Running the tests

In [ ]:

```
# -----------
# Tested Cell
# -----------
# The contents of the cell will be tested by the autograder.
# If they do not pass here, they will not pass there.

# build a fake agent for testing and set some initial conditions
greedy_agent = GreedyAgent()
greedy_agent.q_values = [0, 0, 1.0, 0, 0]
greedy_agent.arm_count = [0, 1, 0, 0, 0]
greedy_agent.last_action = 1

# take a fake agent step
action = greedy_agent.agent_step(reward=1)

# make sure agent took greedy action
assert action == 2

# make sure q_values were updated correctly
assert greedy_agent.q_values == [0, 0.5, 1.0, 0, 0]

# take another step
action = greedy_agent.agent_step(reward=2)
assert action == 2
assert greedy_agent.q_values == [0, 0.5, 2.0, 0, 0]
```

Running the tests

To recap:

* **Always** run every test cell
* Clear your runtime and execute everything once from scratch before submit
* If a test does not pass here, it won’t pass there
* There are no hidden tests
* btw, last I checked (recently) the answers on the internet are outdated and wrong
* If the system is being annoying, chat with us during office hours or class (I’m quite experienced at dealing with Coursera’s …. choices)