CSE 4618

Artificial Intelligence Lab

SWE 20

CSE
IUT

Contents

Lab 0 Python/Autograder Tutorial

The lab tasks for this class assume you use a Linux Distro, e.g. Ubuntu. If you use anything else, we believe you are cool enough to find a workaround, if needed.
Lab 0 will cover the following:

- A mini Linux tutorial.
- Instructions on how to set up the right Python version.
- A mini Python tutorial.
- Lab grading: Every lab release includes its autograder that you can run locally to debug.

Files to Edit: You will fill in portions of `addition.py`, `buyLotsOfFruit.py`, and `shopSmart.py` in `tutorial.zip` during the task. Please do not change the other files in this distribution.
Evaluation: Your code will be autograded for technical correctness. Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation – not the autograder's judgments – will be the final judge of your score. We will review and grade assignments individually to ensure that you receive due credit for your work.
Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please do not try. We trust you all to submit your own work only; please do not let us down. If you do, we will pursue the strongest consequences available to us.
Getting Help: You are not alone! If you find yourself stuck on something, contact us for help. Office hours, Google Classroom, and Emails are there for your support; please use them. We want these labs to be rewarding and instructional, not frustrating and demoralizing. But, we do not know when or how to help unless you ask.
Google Classroom: Please be careful not to post spoilers.
Report Due: 2 weeks after the lab. Please check the submission deadline in the post for more details.

1 Linux Basics

Here are some basic commands to navigate in Linux and edit files.

1.1 File/Directory Manipulation

When you open a terminal window, you are placed at a command prompt:

```
[user@linux: ~]$
```

The prompt shows your username, the host you are logged onto, and your current location in the directory structure (your path). The tilde character is shorthand for your home directory. Note your prompt may look slightly different. To make a directory, use the `mkdir` command. Use `cd` to change to that directory:

```
[user@linux: ~]$ mkdir foo
[user@linux: ~]$ cd foo
[user@linux: ~/foo]$
```

Use `ls` to see a listing of the contents of a directory and `touch` to create an empty file:

```
[user@linux: ~/foo]$ ls
[user@linux: ~/foo]$ touch hello_world
[user@linux: ~/foo]$ ls
hello_world
[user@linux: ~/foo]$ cd ..
[user@linux: ~]$
```

Download `tutorial.zip` into your home directory. Use `unzip` to extract the contents of the zip file:

```
[user@linux: ~/foo]$ ls *.zip
tutorial.zip
[user@linux: ~]$ unzip tutorial.zip
[user@linux: ~]$ cd tutorial
[user@linux: ~/tutorial]$ ls
foreach.py
helloWorld.py
listcomp.py
listcomp2.py
quickSort.py
shop.py
shopTest.py
```

Some other useful Linux commands:

- `cp` copies a file or files
- `rm` removes (deletes) a file
- `mv` moves a file (i.e., cut/paste instead of copy/paste)
- `man` displays documentation for a command
- `pwd` prints your current path
- `xterm` opens a new terminal window
- `firefox` opens a web browser
- Press "Ctrl-c" to kill a running process
- Append & to a command to run it in the background
- `fg` brings a program running in the background to the foreground
- `rmdir` allows you to delete specific folder(s) from your system
- `tar` for archiving files and extracting them
- `mount` to mount folder
- `clear` to clear your current terminal
- `locate` for finding the location of a specific file
- `wget` for downloading files from the web from the terminal
- `history` shows previously used commands

When typing a command, you can press the *Tab* button on your keyboard to autofill what you are typing. For example, let's assume your current working directory contains a folder named "Documents" and you want to navigate to that folder. You can type `cd Documents` and press *Enter* on your keyboard like a peasant! Or you can type a portion of the name of the directory, for example, `cd Docu`, then hit the *Tab* key. The terminal will fill up the rest showing you `cd Documents`. Then you can press *Enter*.

1.2 IDE

To edit Python files, there are lots of IDE and editors. We use Visual Studio Code for the demonstrations. You can use whichever you prefer.

2 Python Installation

We will require Python 3.6 for this lab. Many of you may not have Python 3.6 already installed on your computers. You might have some other version installed. Conda is an easy way to manage many different environments, each with its own Python versions and dependencies. This allows us to avoid conflicts between your preferred version and that required for this course. If you do not have it already, please install Anaconda following the instructions from the link. We will walk through how to set up and use a conda environment.

2.1 Creating a Conda Environment

The command for creating a conda environment with Python 3.6 is:

```
conda create --name <env-name> python=3.6
```

For us, we decide to name our environment `cse4618`, so we run the following command in Command Prompt and then follow the instructions to install any missing packages.

```
[user@linux ~/tutorial]$ conda create --name cse4618 python=3.6
```

2.2 Entering the Environment

To enter the conda environment that we just created, do the following. Note that the Python version within the environment is 3.6, just what we want.

```
[user@linux ~/python_basics]$ conda activate cse4618
(cse4618) [user@linux ~/python_basics]$ python -V
Python 3.6.13 :: Anaconda, Inc.
```

Note: the tag `(<env-name>)` shows you the name of the conda environment that is active.

2.3 Leaving the Environment

Leaving the environment is just as easy.

```
(cse4618) [user@linux ~/python_basics]$ conda deactivate
```

If you check the Python version now, you wll see that the version has now returned to whatever the system default is! Before Conda 4.4, `source activate` and `source deactivate` were preferred for activating and deactivating environments.

3 Python Basics

The programming assignments in this course will be written in Python, an interpreted, object-oriented language that shares some features with both Java and Scheme. This tutorial will walk through the primary syntactic constructions in Python, using short examples.
We encourage you to type all Python codes shown onto your own machine. Make sure it responds the same way.
You may find the Troubleshooting (see below) subsection helpful if you run into problems. It contains a list of the frequent problems Python beginners encounter.

3.1 Required Files

You can download all of the files associated with the Python mini-tutorial as a zip archive: `tutorial.zip`. If you did the Linux tutorial in the previous section, you have already downloaded and unzipped this file.

3.2 Invoking the Interpreter

Python can be run in one of two modes. It can either be used *interactively*, via an interpreter, or it can be called from the command line to execute a *script*. We will first use the Python interpreter interactively.
You invoke the interpreter by entering `python` at the Linux command prompt.

```
(cse4618) [user@linux ~]$ python
Python 3.6.13 |Anaconda, Inc.| (default, Jun  4 2021, 14:25:59)
[GCC 7.5.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

3.3 Operators

The Python interpreter can be used to evaluate expressions, for example, simple arithmetic expressions. If you enter such expressions at the prompt (>>>) they will be evaluated and the result will be returned on the next line.

```
>>> 1 + 1
2
>>> 2 * 3
6
```

Boolean operators also exist in Python to manipulate the primitive `True` and `False` values.

```
>>> 1==0
False
>>> not (1==0)
True
>>> (2==2) and (2==3)
False
>>> (2==2) or (2==3)
True
```

3.4 Strings

Like Java, Python has a built-in string type. The + operator is overloaded to do string concatenation on string values.

```
>>> 'artificial' + "intelligence"
'artificialintelligence'
```

There are many built-in methods that allow you to manipulate strings.

```
>>> 'artificial'.upper()
'ARTIFICIAL'
>>> 'HELP'.lower()
'help'
>>> len('Help')
4
```

You can use `swapcase()` to invert the case of all letters in the string. Notice that we can use either single quotes ‘ ’ or double quotes “ ” to surround the string. This allows for easy nesting of strings.
We can also store expressions into variables.

```
>>> s = 'hello world'
>>> print(s)
hello world
```

```
>>> s.upper()
'HELLO WORLD'
>>> len(s.upper())
11
>>> num = 8.0
>>> num += 2.5
>>> print(num)
10.5
```

In Python, you do not have to declare variables before you assign them.

3.5 Exercise: Dir and Help

Learn about the methods Python provides for strings. To see what methods Python provides for a datatype, use the `dir` and `help` commands:

```
>>> s = 'abc'

>>> dir(s)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getslice__',
'__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__str__', 'capitalize', 'center',
'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper',
'join', 'ljust', 'lower', 'lstrip', 'replace', 'rfind', 'rindex', 'rjust',
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']

>>> help(s.find)
Help on built-in function find:

find(...) method of builtins.str instance
S.find(sub[, start[, end]]) -> int

Return the lowest index in S where the substring sub is found,
such that sub is contained within S[start:end].  Optional
arguments start and end are interpreted as in slice notation.

Return -1 on failure.


>>> s.find('b')
1
```

Press 'q' to back out of a help screen. Try out some of the string functions listed in `dir` (ignore those with underscores '_' around the method name; these are private helper methods.).

3.6 Built-in Data Structures

Python comes equipped with some useful built-in data structures, broadly similar to Java's collections package.

Lists

Lists store a sequence of mutable items:

```
>>> fruits = ['apple', 'orange', 'pear', 'banana']
>>> fruits[0]
'apple'
```

We can use the '+' operator to do list concatenation:

```
>>> otherFruits = ['kiwi', 'strawberry']
>>> fruits + otherFruits
>>> ['apple', 'orange', 'pear', 'banana', 'kiwi', 'strawberry']
```

Python also allows negative indexing from the back of the list. For instance, `fruits[-1]` will access the last element `'banana'`:

```
>>> fruits[-2]
'pear'
>>> fruits.pop()
'banana'
>>> fruits
['apple', 'orange', 'pear']
>>> fruits.append('grapefruit')
>>> fruits
['apple', 'orange', 'pear', 'grapefruit']
>>> fruits[-1] = 'pineapple'
>>> fruits
['apple', 'orange', 'pear', 'pineapple']
```

We can also index multiple adjacent elements using the slice operator. For instance, `fruits[1:3]`, returns a list containing the elements at positions 1 and 2. In general `fruits[start:stop]` will get the elements in `start`, `start+1`, ..., `stop-1`. We can also do `fruits[start:]` which returns all elements starting from the `start` index. Also `fruits[:end]` will return all elements before the element at position `end`:

```
>>> fruits[0:2]
['apple', 'orange']
>>> fruits[:3]
['apple', 'orange', 'pear']
>>> fruits[2:]
['pear', 'pineapple']
>>> len(fruits)
4
```

The items stored in lists can be any Python data type. So for instance we can have lists or lists:

```
>>> lstOfLsts = [['a', 'b', 'c'], [1, 2, 3], ['one', 'two', 'three']]
>>> lstOfLsts[1][2]
3
>>> lstOfLsts[0].pop()
'c'
>>> lstOfLsts
[['a', 'b'], [1, 2, 3], ['one', 'two', 'three']]
```

Other common list operations are:

- Length: `len([1, 2, 3])`
- Repetition: `['hello'] * 4`

- Membership: `3 in [1, 2, 3]`
- Iteration: `for x in [1, 2, 3]:`

3.7 Exercise: Lists

Play with some of the list functions. You can find the methods you can call on an object via the `dir` and get information about them via the `help` command:

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__delslice__', '__doc__', '__eq__', '__ge__', '__getattribute__',
'__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__',
'__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__setslice__', '__str__',
'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
>>> help(list.reverse)
Help on built-in function reverse:

reverse(...)
L.reverse() -- reverse \*IN PLACE\*
>>> lst = ['a', 'b', 'c']
>>> lst.reverse()
>>> ['c', 'b', 'a']`
```

Tuples

A data structure similar to the list is the *tuple*, which is like a list except that it is immutable once it is created (i.e. you cannot change its content once created). Note that tuples are surrounded with parentheses while lists have square brackets.

```
>>> pair = (3, 5)
>>> pair[0]
3
>>> x, y = pair
>>> x
3
>>> y
5
>>> pair[1] = 6
TypeError: object does not support item assignment
```

The attempt to modify an immutable structure raised an exception. Exceptions indicate errors: index out-of-bounds errors, type errors, and so on will all report exceptions in this way.

Sets

A *set* is another data structure that serves as an unordered and non-indexed list with no duplicate items. Below, we show how to create a set:

```
>>> shapes = ['circle', 'square', 'triangle', 'circle']
>>> setOfShapes = set(shapes)
```

Another way of creating a set is shown below:

```
>>> setOfShapes = {'circle', 'square', 'triangle', 'circle'}
```

Next, we show how to add things to the set, test if an item is in the set, and perform common set operations (difference, intersection, union):

```
>>> setOfShapes
set(['circle', 'square', 'triangle'])
>>> setOfShapes.add('polygon')
>>> setOfShapes
set(['circle', 'square', 'triangle', 'polygon'])
>>> 'circle' in setOfShapes
True
>>> 'rhombus' in setOfShapes
False
>>> favoriteShapes = ['circle', 'triangle', 'hexagon']
>>> setOfFavoriteShapes = set(favoriteShapes)
>>> setOfShapes - setOfFavoriteShapes
set(['square', 'polygon'])
>>> setOfShapes & setOfFavoriteShapes
set(['circle', 'triangle'])
>>> setOfShapes | setOfFavoriteShapes
set(['circle', 'square', 'triangle', 'polygon', 'hexagon'])
```

Note that the objects in the set are unordered; you cannot assume that their traversal or print order will be the same across machines!

Dictionaries

The last built-in data structure is the *dictionary* which stores a map from one type of object (the key) to another (the value). The key must be an immutable type (string, number, or tuple). The value can be any Python data type.
Note: In the example below, the printed order of the keys returned by Python could be different than shown below. The reason is that, unlike lists that have a fixed ordering, a dictionary is simply a hash table for which there is no fixed ordering of the keys (like HashMaps in Java). The order of the keys depends on how exactly the hashing algorithm maps keys to buckets, and will usually seem arbitrary. Your code should not rely on key ordering, and you should not be surprised if even a small modification to how your code uses a dictionary results in a new key ordering.

```
>>> studentIds = {'knuth': 42.0, 'turing': 56.0, 'nash': 92.0}
>>> studentIds['turing']
56.0
>>> studentIds['nash'] = 'ninety-two'
>>> studentIds
{'knuth': 42.0, 'turing': 56.0, 'nash': 'ninety-two'}
>>> del studentIds['knuth']
>>> studentIds
{'turing': 56.0, 'nash': 'ninety-two'}
>>> studentIds['knuth'] = [42.0, 'forty-two']
>>> studentIds
{'knuth': [42.0, 'forty-two'], 'turing': 56.0, 'nash': 'ninety-two'}
>>> studentIds.keys()
['knuth', 'turing', 'nash']
>>> studentIds.values()
[[42.0, 'forty-two'], 56.0, 'ninety-two']
>>> studentIds.items()
```

```
[('knuth', [42.0, 'forty-two']), ('turing',56.0), ('nash', 'ninety-two')]
>>> len(studentIds)
3
```

As with nested lists, you can also create nested dictionaries of dictionaries.

3.8 Exercise: Dictionaries

Use `dir` and `help` to learn about the functions you can call on dictionaries.

3.9 Writing Scripts

Now that you have got a handle on using Python interactively, let's write a simple Python script that demonstrates Python's `for` loop. Open the file called `foreach.py`, which should contain the following code:

```
# This is what a comment looks like
fruits = ['apples', 'oranges', 'pears', 'bananas']
for fruit in fruits:
    print(fruit + ' for sale')

fruitPrices = {'apples': 2.00, 'oranges': 1.50, 'pears': 1.75}
for fruit, price in fruitPrices.items():
    if price < 2.00:
        print('%s cost %f a pound' % (fruit, price))
    else:
        print(fruit + ' are too expensive!')
```

At the command line, use the following command in the directory containing `foreach.py`:

```
[(cse4618) user@linux ~/tutorial]$ python foreach.py
apples for sale
oranges for sale
pears for sale
bananas for sale
apples are too expensive!
oranges cost 1.500000 a pound
pears cost 1.750000 a pound
```

Remember that the print statements listing the costs may be in a different order on your screen than in this tutorial; that's because we are looping over unordered dictionary keys. To learn more about control structures (e.g., `if` and `else`) in Python, check out the official Python tutorial section on this topic.

If you like functional programming, you might also like `map` and `filter`:

```
>>> list(map(lambda x: x * x, [1, 2, 3]))
[1, 4, 9]
>>> list(filter(lambda x: x > 3, [1, 2, 3, 4, 5, 4, 3, 2, 1]))
[4, 5, 4]
```

The next snippet of code demonstrates Python's *list comprehension* constructions:

```
nums = [1, 2, 3, 4, 5, 6]
plusOneNums = [x + 1 for x in nums]
oddNums = [x for x in nums if x % 2 == 1]
print(oddNums)
oddNumsPlusOne = [x + 1 for x in nums if x % 2 == 1]
print(oddNumsPlusOne)
```

This code is in a file called `listcomp.py`, which you can run:

```
[(cse4618) user@linux ~/tutorial]$ python listcomp.py
[1, 3, 5]
[2, 4, 6]
```

3.10 Exercise: List Comprehensions

Write a list comprehension which, from a list, generates a lowercase version of each string that has a length greater than five. You can find the solution in `listcomp2.py`.

3.11 Beware of Indentation!

Unlike many other languages, Python uses indentation in the source code for interpretation. So for instance, for the following script:

```
if 0 == 1:
    print('We are in a world of arithmetic pain')
print('Thank you for playing')
```

will output: `Thank you for playing`
But if we had written the script as

```
if 0 == 1:
    print('We are in a world of arithmetic pain')
    print('Thank you for playing')
```

there would be no output. The moral of the story: be careful how you indent! It is best to use four spaces for indentation - that is what the codes in this lab use.

3.12 Tabs vs Spaces

Because Python uses indentation for code evaluation, it needs to keep track of the level of indentation across code blocks. This means that if your Python file switches from using tabs as indentation to spaces as indentation, the Python interpreter will not be able to resolve the ambiguity of the indentation level and throw an exception. Even though the code can be lined up visually in your text editor, Python "sees" a change in the indentation and most likely will throw an exception (or rarely, produce unexpected behavior).
This most commonly happens when opening up a Python file that uses an indentation scheme that is opposite from what your text editor uses (aka, your text editor uses spaces and the file uses tabs). When you write new lines in a code block, there will be a mix of tabs and spaces, even though the whitespace is aligned. Python 3 does not allow mixing tabs and spaces for indentation. For editing the provided codes, using 4 spaces is recommended.

3.13 Writing Functions

As in Java, in Python you can define your own functions:

```
fruitPrices = {'apples':2.00, 'oranges': 1.50, 'pears': 1.75}

def buyFruit(fruit, numPounds):
    if fruit not in fruitPrices:
        print("Sorry we don't have %s" % (fruit))
    else:
        cost = fruitPrices[fruit] * numPounds
        print("That'll be %f please" % (cost))
```

```
# Main Function
if __name__ == '__main__':
    buyFruit('apples',2.4)
    buyFruit('coconuts',2)
```

Rather than having a `main` function as a `main` function as in Java, the `__name__ == '__main__'` check is used to delimit expressions which are executed when the file is called as a script from the command line. The code after the main check is thus the same sort of code you would put in a `main` function in Java.
Save this script as `fruit.py` and run it:

```
(cse4618) [user@linux ~]$ python fruit.py
That'll be 4.800000 please
Sorry we don't have coconuts
```

3.14 Advanced Exercise

Write a `quickSort` function in Python using list comprehensions. Use the first element as the pivot. You can find the solution in `quickSort.py`.

Object Basics

Although this is not a class in object-oriented programming, you will have to use some objects in the lab tasks, so it is worth covering the basics of objects in Python. An object encapsulates data and provides functions for interacting with that data.

Defining Classes

Classes are user-defined prototypes for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation. Here's an example of defining a class named `FruitShop`:

```
class FruitShop:

    def __init__(self, name, fruitPrices):
        """
        name: Name of the fruit shop

        fruitPrices: Dictionary with keys as fruit
        strings and prices for values e.g.
        {'apples': 2.00, 'oranges': 1.50, 'pears': 1.75}
        """
        self.fruitPrices = fruitPrices
        self.name = name
        print('Welcome to %s fruit shop' % (name))

    def getCostPerPound(self, fruit):
        """
        fruit: Fruit string
        Returns cost of 'fruit', assuming 'fruit'
        is in our inventory or None otherwise
        """
        if fruit not in self.fruitPrices:
```

```
            return None
        return self.fruitPrices[fruit]

    def getPriceOfOrder(self, orderList):
        """
        orderList: List of (fruit, numPounds) tuples

        Returns cost of orderList, only including the values of
        fruits that this fruit shop has.
        """
        totalCost = 0.0
        for fruit, numPounds in orderList:
            costPerPound = self.getCostPerPound(fruit)
            if costPerPound != None:
                totalCost += numPounds * costPerPound
        return totalCost

    def getName(self):
        return self.name
```

The `FruitShop` class has some data, the name of the shop, and the prices per pound of some fruit, and it provides functions, or methods, on this data. What advantage is there to wrapping this data in a class?

1. Encapsulating the data prevents it from being altered or used inappropriately,
2. The abstraction that objects provide makes it easier to write general-purpose code.

Using Objects

So how do we make an object and use it? Make sure you have the `FruitShop` implementation in `shop.py`. We then import the code from this file (making it accessible to other scripts) using `import shop`, since `shop.py` is the name of the file. Then, we can create `FruitShop` objects as follows:

```
import shop

shopName = 'Shwapno Express'
fruitPrices = {'apples': 1.00, 'oranges': 1.50, 'pears': 1.75}
shwapnoShop = shop.FruitShop(shopName, fruitPrices)
applePrice = shwapnoShop.getCostPerPound('apples')
print(applePrice)
print('Apples cost $%.2f at %s.' % (applePrice, shopName))

otherName = 'Agora'
otherFruitPrices = {'kiwis':6.00, 'apples': 4.50, 'peaches': 8.75}
otherFruitShop = shop.FruitShop(otherName, otherFruitPrices)
otherPrice = otherFruitShop.getCostPerPound('apples')
print(otherPrice)
print('Apples cost $%.2f at %s.' % (otherPrice, otherName))
print("My, that's expensive!")
```

This code is in `shopTest.py`; you can run it like this:

```
(cse4618) [user@linux ~/tutorial]$ python shopTest.py
Welcome to Shwapno Express fruit shop
1.0
Apples cost $1.00 at Shwapno Express.
```

```
Welcome to Agora fruit shop
4.5
Apples cost $4.50 at Agora.
My, that's expensive!
```

So what just happened? The `import shop` statement told Python to load all of the functions and classes in `shop.py`. The line `shwapnoShop = shop.FruitShop(shopName, fruitPrices)` constructs an *instance* of the `FruitShop` class defined in *shop.py*, by calling the `__init__` function in that class. Note that we only passed two arguments in, while `__init__` seems to take three arguments: `(self, name, fruitPrices)`. The reason for this is that all methods in a class have `self` as the first argument. The `self` variable contains all the data (`name` and `fruitPrices`) for the current specific instance (similar to `this` in Java). The print statements use the substitution operator (described in the Python docs if you are curious).

Static vs Instance Variables

If the value of a variable varies from object to object, then such variables are called instance variables. For every object, a separate copy of the instance variable will be created. If the value of a variable is not varied from object to object, such types of variables we have to declare within the class directly but outside of methods. Such types of variables are called Static variables. For the entire class, only one copy of the static variable will be created and shared by all objects of that class. We can access static variables either by class name or by object reference. But it is recommended to use the class name.

The following example illustrates how to use static and instance variables in Python:

Create the `person_class.py` containing the following code:

```
class Person:
    population = 0

    def __init__(self, myAge):
        self.age = myAge
        Person.population += 1

    def get_population(self):
        return Person.population

    def get_age(self):
        return self.age
```

We first compile the script:

```
(cse4618) [user@linux ~]$ python person_class.py
```

Now use the class as follows:

```
>>> import person_class
>>> p1 = person_class.Person(12)
>>> p1.get_population()
1
>>> p2 = person_class.Person(63)
>>> p1.get_population()
2
>>> p2.get_population()
2
>>> p1.get_age()
```

```
12
>>> p2.get_age()
63
```

In the code above, `age` is an instance variable and `population` is a static variable. `population` is shared by all instances of the `Person` class whereas each instance has its own `age` variable.

Modules

A file containing a group of Functions, Classes, and Variables that you want to include in your applications is called a Module. You can include whatever file you want in your code as a module. However, that file must be saved using '.py' extension. The benefit of using modules is that, once you have defined any function in a module and saved it, you can use the `import` statement to import it from any other source file. For aliasing a module, we use `as` keyword.

3.15 More Python Tips and Tricks

This tutorial has briefly touched on some major aspects of Python that will be relevant to the course. Here are some more useful tidbits:

- Use `range` to generate a sequence of integers, useful for generating traditional indexed `for` loops:

```
for index in range(3):
    print(index)
```

- After importing a file, if you edit a source file, the changes will not be immediately propagated in the interpreter. For this, use the `reload` command:

```
>>> reload(shop)
```

- Swap two variables with one line of code.

```
>>> a = 7
>>> b = 5
>>> b, a = a, b
>>> a
5
>>> b
7
```

- Assign multiple values at the same time:

```
>>> a = [1, 2, 3]
>>> x, y, z = a
>>> x
1
>>> y
2
>>> z
3
```

3.16 Troubleshooting

These are some problems (and their solutions) that new Python learners commonly encounter.

- **Problem:**

  ImportError: No module named py

Solution:

When using `import`, do not include the ".py" from the filename.

For example, you should say: `import shop`

NOT: `import shop.py`

▹ **Problem:**

NameError: name 'MY VARIABLE' is not defined

Even after importing you may see this.

Solution:

To access a member of a module, you have to type `MODULE NAME.MEMBER NAME`, where `MODULE NAME` is the name of the `.py` file, and `MEMBER NAME` is the name of the variable (or function) you are trying to access.

Another reason for this problem would be using due to scope issues. It is best to stick with local variables within the code block. If you want to utilize certain variables throughout the program, use the global variable format.

▹ **Problem:**

TypeError: 'dict' object is not callable

Solution:

Dictionary look ups are done using square brackets: [and]. NOT parenthesis: (and).

▹ **Problem:**

ValueError: too many values to unpack

Solution:

Make sure the number of variables you are assigning in a `for` loop matches the number of elements in each item of the list. Similarly for working with tuples.

For example, if `pair` is a tuple of of two elements (e.g `pair =('apple', 2.0)`) then the following code would cause the "too many values to unpack error":

```
(a, b, c) = pair
```

Here is a problematic scenario involving a `for` loop:

```
pairList = [('apples', 2.00), ('oranges', 1.50), ('pears', 1.75)]
for fruit, price, color in pairList:
    print('%s fruit costs %f and is the color %s' % (fruit, price, color))
```

▹ **Problem:**

AttributeError: 'list' object has no attribute 'length' (or something similar)

Solution:

Finding length of lists is done using `len(NAME OF LIST)`.

▹ **Problem:**

Changes to a file are not taking effect.

Solution:

1. Make sure you are saving all your files after any changes.
2. If you are editing a file in a window different from the one you are using to execute python, make sure you `reload(YOUR_MODULE)` to guarantee your changes are being reflected. `reload` works similarly to `import`

3.17 More References

▹ The place to go for more Python information: www.python.org

▹ A good reference book: Learning Python

4 Autograding

To get you familiarized with the autograder, we will ask you to code, test, and submit solutions for three questions. You can download all of the files associated with the autograder tutorial as a zip archive: `tutorial.zip`. If not done so already, download the file and unzip it:

```
[user@linux ~]$ unzip tutorial.zip
[user@linux ~]$ cd tutorial
[user@linux ~/tutorial]$ ls
addition.py
autograder.py
buyLotsOfFruit.py
grading.py
projectParams.py
shop.py
shopSmart.py
testClasses.py
testParser.py
test_cases
tutorialTestClasses.py
```

This contains several files you will edit or run:

- `adition.py`: source file for question 1
- `buyLotsOfFruit.py`: source file for question 2
- `shop.py`: source file for question 3
- `shopSmart.py`: source file for question 3
- `autograder.py`: autograding script (see below)

and others you can ignore:

- `test_cases`: directory contains the test cases for each question
- `grading.py`: autograder code
- `testClasses.py`: autograder code
- `tutorialTestClasses.py`: test classes for this particular project
- `projectParams.py`: project parameters

The command `python autograder.py` grades your solution to all three problems. If we run it before editing any files we get a page or two of output:

```
(cse4618) [user@linux ~/tutorial]$ python autograder.py
Starting on 1-21 at 23:39:51

Question q1
===========
*** FAIL: test_cases/q1/addition1.test
***     add(a, b) must return the sum of a and b
***     student result: "0"
***     correct result: "2"
*** FAIL: test_cases/q1/addition2.test
***     add(a, b) must return the sum of a and b
***     student result: "0"
***     correct result: "5"
*** FAIL: test_cases/q1/addition3.test
```

```
***     add(a, b) must return the sum of a and b
***     student result: "0"
***     correct result: "7.9"
*** Tests failed.

### Question q1: 0/1 ###


Question q2
===========
*** FAIL: test_cases/q2/food_price1.test
***     buyLotsOfFruit must compute the correct cost of the order
***     student result: "0.0"
***     correct result: "12.25"
*** FAIL: test_cases/q2/food_price2.test
***     buyLotsOfFruit must compute the correct cost of the order
***     student result: "0.0"
***     correct result: "14.75"
*** FAIL: test_cases/q2/food_price3.test
***     buyLotsOfFruit must compute the correct cost of the order
***     student result: "0.0"
***     correct result: "6.4375"
*** Tests failed.

### Question q2: 0/1 ###


Question q3
===========
Welcome to shop1 fruit shop
Welcome to shop2 fruit shop
*** FAIL: test_cases/q3/select_shop1.test
***     shopSmart(order, shops) must select the cheapest shop
***     student result: "None"
***     correct result: "<FruitShop: shop1>"
Welcome to shop1 fruit shop
Welcome to shop2 fruit shop
*** FAIL: test_cases/q3/select_shop2.test
***     shopSmart(order, shops) must select the cheapest shop
***     student result: "None"
***     correct result: "<FruitShop: shop2>"
Welcome to shop1 fruit shop
Welcome to shop2 fruit shop
Welcome to shop3 fruit shop
*** FAIL: test_cases/q3/select_shop3.test
***     shopSmart(order, shops) must select the cheapest shop
***     student result: "None"
***     correct result: "<FruitShop: shop3>"
*** Tests failed.

### Question q3: 0/1 ###
```

```
Finished at 23:39:51

Provisional grades
==================
Question q1: 0/1
Question q2: 0/1
Question q3: 0/1
------------------
Total: 0/3

Your grades are NOT yet registered.  To register your grades, make sure to follow
your instructor's guidelines to receive credit on your project.
```

For each of the three questions, this shows the results of that question's tests, the question's grade, and a final summary at the end. Because you have not yet solved the questions, all the tests fail. As you solve each question you may find some tests pass while others fail. When all tests pass for a question, you get full marks.
Looking at the results for question 1, you can see that it has failed three tests with the error message "add(a,b) must return the sum of a and b". The answer your code gives is always 0, but the correct answer is different. We will fix that in the next section.

5 Question 1: Addition

Open `addition.py` and look at the definition of `add`:

```
def add(a, b):
    "Return the sum of a and b"
    "*** YOUR CODE HERE ***"
    return 0
```

The tests called this with `a` and `b` set to different values, but the code always returned zero. Modify this definition to read:

```
def add(a, b):
    "Return the sum of a and b"
    print("Passed a=%s and b=%s, returning a+b=%s" % (a,b,a+b))
    return a+b
```

Now rerun the autograder. You will see something like this (omitting the results for questions 2 and 3):

```
(cse4618) [user@linux ~/tutorial]$ python autograder.py -q q1
Starting on 1-21 at 23:52:05

Question q1
===========
Passed a=1 and b=1, returning a+b=2
*** PASS: test_cases/q1/addition1.test
***     add(a,b) returns the sum of a and b
Passed a=2 and b=3, returning a+b=5
*** PASS: test_cases/q1/addition2.test
***     add(a,b) returns the sum of a and b
Passed a=10 and b=-2.1, returning a+b=7.9
*** PASS: test_cases/q1/addition3.test
***     add(a,b) returns the sum of a and b
```

```
### Question q1: 1/1 ###

Finished at 23:41:01

Provisional grades
==================
Question q1: 1/1
Question q2: 0/1
Question q3: 0/1
------------------
Total: 1/3
```

You now pass all tests, getting full marks for question 1. Notice the new lines "Passed a=..." which appear before "*** PASS: ...". These are produced by the print statement in `add`. You can use print statements like that to output information useful for debugging.

6 Question 2: buyLotsOfFruit function

Implement the `buyLotsOfFruit(orderList)` function in `buyLotsOfFruit.py` which takes a list of `(fruit, pound)` tuples and returns the cost of your list. If there is some `fruit` in the list which does not appear in `fruitPrices` it should print an error message and return `None`. Please do not change the `fruitPrices` variable.

Run `python autograder.py` until question 2 passes all tests and you get full marks. Each test will confirm that `buyLotsOfFruit(orderList)` returns the correct answer given various possible inputs. For example, `test_cases/q2/food_price1.test` tests whether:

```
Cost of [('apples', 2.0), ('pears', 3.0), ('limes', 4.0)] is 12.25
```

7 Question 3: shopSmart function

Fill in the function `shopSmart(orders,shops)` in `shopSmart.py`, which takes an `orderList` (like the kind passed in to `FruitShop.getPriceOfOrder`) and a list of `FruitShop` and returns the `FruitShop` where your order costs the least amount in total. Don't change the file name or variable names, please. Note that we will provide the `shop.py` implementation as a "support" file, so you don't need to submit yours.

Run python `autograder.py` until question 3 passes all tests and you get full marks. Each test will confirm that `shopSmart(orders, shops)` returns the correct answer given various possible inputs. For example, with the following variable definitions:

```
orders1 = [('apples',1.0), ('oranges',3.0)]
orders2 = [('apples',3.0)]
dir1 = {'apples': 2.0, 'oranges':1.0}
shop1 =  shop.FruitShop('shop1',dir1)
dir2 = {'apples': 1.0, 'oranges': 5.0}
shop2 = shop.FruitShop('shop2',dir2)
shops = [shop1, shop2]
```

`test_cases/q3/select_shop1.test` tests whether:

```
shopSmart.shopSmart(orders1, shops) == shop1
```

and `test_cases/q3/select_shop2.test` tests whether:

```
shopSmart.shopSmart(orders2, shops) == shop2
```

8 Evaluation

Once you are done with the tasks, call your course teacher and show them the autograder results and codes.

9 Submission

Submit one file: `StudentID_L0.pdf` (StudentID will be replaced by your student ID) under **Assignment 0** on **Google Classroom**. The report can contain (but is not limited to) **introduction** (briefly introduce the problem you tackled, highlight the specific algorithm/technique you implemented), **problem analysis** (discuss the nuances of the problem, explain what modification or unique aspects you were asked to introduce), **solution explanation** (provide an overview of your approach, include portions of the code only where necessary for clarification, highlight key decisions you made during the implementation, analyze the complexity of your code), **findings and insights** (share any interesting observations/insights you gained, discuss how your solution performed in various scenarios), **challenges faced** (describe any challenges you encountered, explain how you overcame these challenges), **hyperparameter exploration** (discuss the impact of different hyperparameter values on your solution's behavior, analyze how changes influenced the performance), **additional information** (feel free to include any extra details, such as extensions, additional features, or creative elements you added to your solution). The report is about insights, analysis, and reflection — no need to duplicate your entire code. Be clear and concise in your explanations. This is your chance to showcase your understanding and creativity, so make the most of it.

You will have **2 weeks** to submit the file. Plagiarism is strictly prohibited and will result in significant penalties.

Lab 1 Uninformed Search

In this lab, your Pacman agent will find paths through his maze world, both to reach a particular location and to collect food efficiently. You will build general search algorithms and apply them to Pacman scenarios.

As in Lab 0, this lab includes an autograder for you to grade your answers on your machine. This can be run with the command:

```
python autograder.py
```

See the autograder tutorial in Lab 0 for more information about using the autograder.

The code for this lab consists of several Python files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore. You can download all the code and supporting files in `usearch.zip`.

Files you will edit:	
`search.py`	Where all of your search algorithms will reside.
Files you might want to look at:	
`searchAgents.py`	Where all of your search-based agents reside.
`pacman.py`	The main file that runs Pacman games. This file describes a Pacman GameState type, which you use in this lab.
`game.py`	The logic behind how the Pacman world works. This file describes several supporting types like AgentState, Agent Direction, and Grid
`util.py`	Useful data structures for implementing search algorithms.
Supporting files you can ignore:	
`graphicsDisplay.py`	Graphics for Pacman
`graphicsUtils.py`	Support for Pacman graphics
`textDisplay.py`	ASCII graphics for Pacman
`ghostAgents.py`	Agents to control ghosts
`keyboardAgents.py`	Keyboard interfaces to control Pacman
`layout.py`	Code for reading layout files and storing their contents
`autograder.py`	Lab autograder
`testParser.py`	Parses autograder test and solution files
`testClasses.py`	General autograding test classes
`test_cases/`	Directory containing the test cases for each question
`searchTestClasses.py`	Lab 1 specific autograding test classes

Files to Edit and Submit: You will fill in portions of `search.py` during the task. Please *do not* change the other files in this distribution or submit any of our original files other than these files.

Evaluation: Your code will be autograded for technical correctness. Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation – not the autograder's judgments – will be the final judge of your score. We will review and grade assignments individually to ensure that you receive due credit for your work.

Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please do not try. We trust you all to submit your own work only; please do not let us down. If you do, we will

pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact us for help. Office hours, Google Classroom, and Emails are there for your support; please use them. We want these labs to be rewarding and instructional, not frustrating and demoralizing. But, we do not know when or how to help unless you ask.

Google Classroom: Please be careful not to post spoilers.

Report Due: 2 weeks after the lab. Please check the submission deadline in the post for more details.

1 Welcome to Pacman

After downloading the code (`usearch.zip`), unzipping it, and changing to the directory, you should be able to play a game of Pacman by typing the following at the command line:

```
python pacman.py
```

Pacman lives in a shiny blue world of twisting corridors and tasty round treats. Navigating this world efficiently will be Pacman's first step in mastering his domain.

The simplest agent in `searchAgents.py` is called the `GoWestAgent`, which always goes West (a trivial reflex agent). The agent can occasionally win:

```
python pacman.py --layout testMaze --pacman GoWestAgent
```

But, things get ugly for this agent when turning is required:

```
python pacman.py --layout tinyMaze --pacman GoWestAgent
```

If Pacman gets stuck, you can exit the game by typing CTRL-c into your terminal.

Soon, your agent will solve not only `tinyMaze`, but any maze you want.

Note that `pacman.py` supports a number of options that can each be expressed in a long way (e.g., `-layout`) or a short way (e.g., `-l`). You can see the list of all options and their default values via:

```
python pacman.py -h
```

Also, all of the commands that appear in this lab also appear in `commands.txt`, for easy copying and pasting. In Linux, you can even run all these commands in order with `bash commands.txt`.

2 Question 1 (3 points): Finding a Fixed Food Dot using Depth First Search

In `searchAgents.py`, you will find a fully implemented `SearchAgent`, which plans out a path through Pacman's world and then executes that path step-by-step. The search algorithms for formulating a plan are not implemented — that's your job.

First, test that the `SearchAgent` is working correctly by running:

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

The command above tells the `SearchAgent` to use `tinyMazeSearch` as its search algorithm, which is implemented in `search.py`. Pacman should navigate the maze successfully.

Now it is time to write full-fledged generic search functions to help Pacman plan routes! The pseudocode for the search algorithms you will write can be found in the lecture slides. Remember that a search node must contain not only a state but also the information necessary to reconstruct the path (plan) which gets to that state.

Important note: All of your search functions need to return a list of *actions* that will lead the agent from the start to the goal. These actions all have to be legal moves (valid directions, no moving through walls).

Important note: Make sure to use the `Stack`, `Queue` and `PriorityQueue` data structures provided to you in `util.py`! These data structure implementations have particular properties which are required for compatibility with the autograder.

Hint: Each algorithm is very similar. Algorithms for DFS, BFS, UCS, and A* differ only in the details of how the fringe is managed. So, concentrate on getting DFS right and the rest should be relatively straightforward. Indeed, one possible implementation requires only a single generic search method which is configured with an algorithm-specific queuing strategy. (Your implementation need not be of this form to receive full credit).

Implement the depth-first search (DFS) algorithm in the `depthFirstSearch` function in `search.py`. The function takes one parameter, `problem`, which provides you with a few important functions, such as `getStartState()` to initialize your algorithm, `isGoalState()` that takes the current state to check whether it is a goal state or not, and `getSuccessors()` that gives you a set of successor *elements* for a given state. Each *element* consists of 3 items: next state (the next position of Pacman), action (the action required to get to the next state from the current state), and cost (the cost of executing the action). For DFS and BFS, where our goal is to find a path to food, we can assume the cost as 1. To make your algorithm complete, write the graph search version of DFS, which avoids expanding any already visited states.

Your code should quickly find a solution for:

```
python pacman.py -l tinyMaze -p SearchAgent
python pacman.py -l mediumMaze -p SearchAgent
python pacman.py -l bigMaze -z .5 -p SearchAgent
```

The Pacman board will show an overlay of the states explored, and the order in which they were explored (brighter red means earlier exploration). Is the exploration order what you would have expected? Does Pacman actually go to all the explored squares on his way to the goal?

Hint: If you use a `Stack` as your data structure, the solution found by your DFS algorithm for `mediumMaze` should have a length of 130 (provided you push successors onto the fringe in the order provided by getSuccessors; you might get 246 if you push them in the reverse order). Is this a least cost solution? If not, think about what depth-first search is doing wrong.

Grading: Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q1
```

3 Question 2 (3 points): Breadth First Search

Implement the breadth-first search (BFS) algorithm in the `breadthFirstSearch` function in `search.py`. Again, write a graph search algorithm that avoids expanding any already visited states. Test your code the same way you did for the depth-first search.

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```

Does BFS find a least cost solution? If not, check your implementation.

Hint: If Pacman moves too slowly for you, try the option `--frameTime 0`.

Note: If you have written your search code generically, your code should work equally well for the eight-puzzle search problem without any changes.

```
python eightpuzzle.py
```

Grading: Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q2
```

4 Question 3 (3 points): Varying the Cost Function

While BFS will find a fewest-actions path to the goal, we might want to find paths that are "best" in other senses. Consider `mediumDottedMaze` and `mediumScaryMaze`.

By changing the cost function, we can encourage Pacman to find different paths. For example, we can charge more for dangerous steps in ghost-ridden areas or less for steps in food-rich areas, and a rational Pacman agent should adjust its behavior in response.

Implement the uniform-cost graph search algorithm in the `uniformCostSearch` function in `search.py`. We encourage you to look through `util.py` for some data structures that may be useful in your implementation. You should now observe successful behavior in all three of the following layouts, where the agents below are all UCS agents that differ only in the cost function they use (the agents and cost functions are written for you):

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

Note: You should get very low and very high path costs for the `StayEastSearchAgent` and `StayWestSearchAgent` respectively, due to their exponential cost functions (see `searchAgents.py` for details).

Grading: Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q3
```

5 Evaluation

Once you are done with the tasks, call your course teacher and show them the autograder results and codes.

6 Submission

Submit one file: `StudentID_L1.pdf` (StudentID will be replaced by your student ID) under **Assignment 1** on **Google Classroom**. The report can contain (but is not limited to) **introduction** (briefly introduce the problem you tackled, highlight the specific algorithm/technique you implemented), **problem analysis** (discuss the nuances of the problem, explain what modification or unique aspects you were asked to introduce), **solution explanation** (provide an overview of your approach, include portions of the code only where necessary for clarification, highlight key decisions you made during the implementation, analyze the complexity of your code), **findings and insights** (share any interesting observations/insights you gained, discuss how your solution performed in various scenarios), **challenges faced** (describe any challenges you encountered, explain how you overcame these challenges), **hyperparameter exploration** (discuss the impact of different hyperparameter values on your solution's behavior, analyze how changes influenced the performance), **additional information** (feel free to include any extra details, such as extensions, additional features, or creative elements you added to your solution). The report is about insights, analysis, and reflection — no need to duplicate your entire code. Be clear and concise in your explanations. This is your chance to showcase your understanding and creativity, so make the most of it.

You will have **2 weeks** to submit the file. Plagiarism is strictly prohibited and will result in significant penalties.

Lab 2 Informed Search

In this lab, your Pacman agent will find paths through his maze world, both to reach a particular location and to collect food efficiently. You will build general search algorithms and apply them to Pacman scenarios.
As in Lab 0, this lab includes an autograder for you to grade your answers on your machine. This can be run with the command:

```
python autograder.py
```

See the autograder tutorial in Lab 0 for more information about using the autograder.
The code for this lab consists of several Python files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore. You can download all the code and supporting files in `isearch.zip`.

Files you will edit:	
`search.py`	Where all of your search algorithms will reside.
`searchAgents.py`	Where all of your search-based agents will reside.
Files you might want to look at:	
`pacman.py`	The main file that runs Pacman games. This file describes a Pacman GameState type, which you use in this lab.
`game.py`	The logic behind how the Pacman world works. This file describes several supporting types like AgentState, Agent Direction, and Grid
`util.py`	Useful data structures for implementing search algorithms.
`commands.txt`	Contains the commands mentioned in the PDF but in a single line
Supporting files you can ignore:	
`graphicsDisplay.py`	Graphics for Pacman
`graphicsUtils.py`	Support for Pacman graphics
`textDisplay.py`	ASCII graphics for Pacman
`ghostAgents.py`	Agents to control ghosts
`keyboardAgents.py`	Keyboard interfaces to control Pacman
`layout.py`	Code for reading layout files and storing their contents
`autograder.py`	Lab autograder
`testParser.py`	Parses autograder test and solution files
`testClasses.py`	General autograding test classes
`test_cases/`	Directory containing the test cases for each question
`searchTestClasses.py`	Task 1 specific autograding test classes

Files to Edit and Submit: You will fill in portions of `search.py` and `searchAgents.py` during the task. Please *do not* change the other files in this distribution or submit any of our original files other than these files.
Evaluation: Your code will be autograded for technical correctness. Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation – not the autograder's judgments – will be the final judge of your score. We will review and grade assignments individually to ensure that you receive due credit for your work.
Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If

you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please do not try. We trust you all to submit your own work only; please do not let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact us for help. Office hours, Google Classroom, and Emails are there for your support; please use them. We want these labs to be rewarding and instructional, not frustrating and demoralizing. But, we do not know when or how to help unless you ask.

Google Classroom: Please be careful not to post spoilers.

Report Due: 2 weeks after the lab. Please check the submission deadline in the post for more details.

1 Welcome (back) to Pacman

After downloading the code (`isearch.zip`), unzipping it, and changing to the directory, you should be able to play a game of Pacman by typing the following at the command line:

```
python pacman.py
```

Pacman lives in a shiny blue world of twisting corridors and tasty round treats. Navigating this world efficiently will be Pacman's first step in mastering his domain.

The simplest agent in `searchAgents.py` is called the `GoWestAgent`, which always goes West (a trivial reflex agent). The agent can occasionally win:

```
python pacman.py --layout testMaze --pacman GoWestAgent
```

But, things get ugly for this agent when turning is required:

```
python pacman.py --layout tinyMaze --pacman GoWestAgent
```

If Pacman gets stuck, you can exit the game by typing CTRL-c into your terminal.

Soon, your agent will solve not only `tinyMaze`, but any maze you want.

Note that `pacman.py` supports a number of options that can each be expressed in a long way (e.g., `-layout`) or a short way (e.g., `-l`). You can see the list of all options and their default values via:

```
python pacman.py -h
```

All of the commands that appear in this lab also appear in `commands.txt`, for easy copying and pasting. In Linux, you can even run all these commands in order with `bash commands.txt`.

2 Question 1 (3 points): A* search

Implement A* graph search in the empty function `aStarSearch` in `search.py`. A* takes a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information). The `nullHeuristic` heuristic function in `search.py` is a trivial example.

You can test your A* implementation on the original problem of finding a path through a maze to a fixed position using the Manhattan distance heuristic (implemented already as `manhattanHeuristic` in `searchAgents.py`).

```
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
```

You should see that A* finds the optimal solution slightly faster than uniform cost search (about 549 vs. 620 search nodes expanded in our implementation, but ties in priority may make your numbers differ slightly). What happens on `openMaze` for the various search strategies?

Grading: Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q1
```

3 Question 2 (3 points): Finding All the Corners

Note: Question 2 requires implementation of `breadthFirstSearch` *function from* `search.py`*, which has been already done for you.*

The real power of A* will only be apparent with a more challenging search problem. Now, it is time to formulate a new problem and design a heuristic for it.

In *corner mazes*, there are four dots, one in each corner. Our new search problem is to find the shortest path through the maze that touches all four corners (whether the maze has food there or not). Note that for some mazes like `tinyCorners`, the shortest path does not always go to the closest food first! *Hint:* the shortest path through `tinyCorners` takes 28 steps.

Implement the `CornersProblem` search problem in `searchAgents.py`. You will need to choose a state representation that encodes all the information necessary to detect whether all four corners have been reached. Now, your search agent should solve:

```
python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
```

To receive full credit, you need to define an abstract state representation that does not encode irrelevant information (like the position of ghosts, where extra food is, etc.). In particular, do not use a Pacman `GameState` as a search state. Your code will be very, very slow if you do (and also wrong).

An instance of the `CornersProblem` class represents an entire search problem, not a particular state. Particular states are returned by the functions you write, and your functions return a data structure of your choosing (e.g. tuple, set, etc.) that represents a state.

Furthermore, while a program is running, remember that many states simultaneously exist, all on the queue of the search algorithm, and they should be independent of each other. In other words, you should not have only one state for the entire `CornersProblem` object; your class should be able to generate many different states to provide to the search algorithm.

Hint 1: The only parts of the game state you need to reference in your implementation are the starting Pacman position and the location of the four corners.

Hint 2: When coding up `getSuccessors`, make sure to add children to your successors list with a cost of 1.

Grading: Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q2
```

4 Question 3 (3 points): Corners Problem: Heuristic

Note: Make sure to complete Question 1 before working on Question 3, because Question 3 builds upon your answer for Question 1.

Implement a non-trivial, consistent heuristic for the `CornersProblem` in `cornersHeuristic`.

```
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

Note: `AStarCornersAgent` is a shortcut for

```
-p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic
```

Admissibility vs. Consistency: Remember, heuristics are just functions that take search states and return numbers that estimate the cost to the nearest goal. More effective heuristics will return values closer to the actual goal costs. To be *admissible*, the heuristic values must be lower bounds on the actual shortest path cost to the nearest goal (and non-negative). To be *consistent*, it must additionally hold that if an action has cost c, then taking that action can only cause a drop in heuristic of at most c.

Remember that admissibility isn't enough to guarantee correctness in graph search – you need a stronger condition of consistency. However, admissible heuristics are usually also consistent, especially if they are derived from problem relaxations. Therefore it is usually easiest to start out by brainstorming admissible heuristics. Once you have an admissible heuristic that works well, you can check whether it is indeed consistent, too. The only way to guarantee consistency is with a proof. However, inconsistency can often be detected by verifying that for each node you expand, its successor nodes are equal or higher in f-value. Moreover, if UCS and A* ever return paths of different lengths, your heuristic is inconsistent. This stuff is tricky!

Non-Trivial Heuristics: The trivial heuristics are the ones that return zero everywhere (UCS) and the heuristic that computes the true completion cost. The former won't save you any time, while the latter will timeout the autograder. You want a heuristic that reduces total compute time, though for this assignment the autograder will only check node counts (aside from enforcing a reasonable time limit).

Grading: Your heuristic must be a non-trivial non-negative consistent heuristic to receive any points. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands, you will be graded:

Number of nodes expanded	Grade
more than 2000	0/3
at most 2000	1/3
at most 1600	2/3
at most 1200	3/3

Remember: If your heuristic is inconsistent, you will receive *no* credit, so be careful!

Grading: Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q3
```

5 Question 4 (4 points): Eating All The Dots

Note: Make sure to complete Question 1 before working on Question 4, because Question 4 builds upon your answer for Question 1.

Now we will solve a hard search problem: eating all the Pacman food in as few steps as possible. For this, we'll need a new search problem definition that formalizes the food-clearing problem: `FoodSearchProblem` in `searchAgents.py` (implemented for you). A solution is defined to be a path that collects all of the food in the Pacman world. For the present lab, solutions do not take into account any ghosts or power pellets; solutions only depend on the placement of walls, regular food, and Pacman. (Of course, ghosts can ruin the execution of a solution! We will get to that in a future lab task in shaa Allah.) If you have written your general search methods correctly, `A*` with a null heuristic (equivalent to uniform-cost search) should quickly find an optimal solution to `testSearch` with no code change on your part (total cost of 7).

```
python pacman.py -l testSearch -p AStarFoodSearchAgent
```

Note: `AStarFoodSearchAgent` is a shortcut for

```
-p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic
```

You should find that UCS starts to slow down even for the seemingly simple `tinySearch`. As a reference, our implementation takes 2.1 seconds to find a path of length 27 after expanding 5057 search nodes.

Fill in `foodHeuristic` in `searchAgents.py` with a *consistent* heuristic for the `FoodSearchProblem`. Try your agent on the `trickySearch` board:

```
python pacman.py -l trickySearch -p AStarFoodSearchAgent
```

Our UCS agent finds the optimal solution in about 16.9 seconds, exploring over 16,000 nodes.
Any non-trivial non-negative consistent heuristic will receive 1 point. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands, you wll get additional points:

Number of nodes expanded	Grade
more than 15000	1/4
at most 15000	2/4
at most 12000	3/4
at most 9000	4/4 (full credit; medium)
at most 7000	5/4 (optional extra credit; hard)

Remember: If your heuristic is inconsistent, you will receive no credit, so be careful! Can you solve `mediumSearch` in a short time? If so, we are either very, very impressed, or your heuristic is inconsistent.
Grading: Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q4
```

6 Question 5 (3 points): Suboptimal Search

Sometimes, even with A* and a good heuristic, finding the optimal path through all the dots is hard. In these cases, we woud still like to find a reasonably good path, quickly. In this section, you will write an agent that always greedily eats the closest dot. `ClosestDotSearchAgent` is implemented for you in `searchAgents.py`, but it is missing a key function that finds a path to the closest dot.
Implement the function `findPathToClosestDot` in `searchAgents.py`. Notice that, the function calls `AnyFoodSearch` class, which is also missing a goal test function. You need to implement that as well. Our agent solves this maze (suboptimally!) in under a second with a path cost of 350:

```
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
```

Your `ClosestDotSearchAgent` won't always find the shortest possible path through the maze. Make sure you understand why and try to come up with a small example where repeatedly going to the closest dot does not result in finding the shortest path for eating all the dots.
Grading: Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q5
```

7 Evaluation

Once you are done with the tasks, call your course teacher and show them the autograder results and codes.

8 Submission

Submit one file: `StudentID_L2.pdf` (StudentID will be replaced by your student ID) under **Assignment 2** on **Google Classroom**. The report can contain (but is not limited to) **introduction** (briefly introduce the problem you tackled, highlight the specific algorithm/technique you implemented), **problem analysis** (discuss the nuances of the problem, explain what modification or unique aspects you were asked to introduce), **solution explanation** (provide an overview of your approach, include portions of the code only where necessary for clarification, highlight key decisions you made during the implementation, analyze the complexity of your code), **findings and insights** (share any interesting observations/insights you gained, discuss how your solution performed in various scenarios), **challenges faced** (describe any challenges

you encountered, explain how you overcame these challenges), **hyperparameter exploration** (discuss the impact of different hyperparameter values on your solution's behavior, analyze how changes influenced the performance), **additional information** (feel free to include any extra details, such as extensions, additional features, or creative elements you added to your solution). The report is about insights, analysis, and reflection — no need to duplicate your entire code. Be clear and concise in your explanations. This is your chance to showcase your understanding and creativity, so make the most of it.

You will have **2 weeks** to submit the file. Plagiarism is strictly prohibited and will result in significant penalties.

Lab 3 Constraint Satisfaction Problem

In this lab, you will create simple CSPs and solve them using the amazing applet provided by Alspace.
The code for this lab requires running a Java applet. So you need to have Java installed on your computer. To check if Java is installed on your computer, run the following command in the terminal:

```
java -version
```

The output should display the version of the Java package installed on your system. If it does not, you need to install Java Runtime Environment (JRE).

Files to Edit and Submit: You will create XML files corresponding to the solutions of the tasks provided. You will load them one by one to demonstrate whether they work or not.

Evaluation: Your implementations will be inspected manually.

Academic Dishonesty: We will be checking your implementation against other submissions in the class for logical redundancy. If you copy someone else's files and submit it with minor changes, we will know. We trust you all to show your own work only; please don't let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact us for help. Office hours, Google Classroom, and Emails are there for your support; please use them. We want these labs to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

Google Classroom: Please be careful not to post spoilers.

Report Due: 2 weeks after the lab. Please check the submission deadline in the post for more details.

1 Welcome to Consistency Based CSP Solver

After downloading the applet (`constraint.jar`) and navigating to the appropriate directory, you can open it using the following command:

```
java -jar constraint.jar
```

After opening it, you will be able to load sample CSPs by going to `File` -> `Load Sample CSP` and choosing the one that you want to load. For example, we have already seen the `Five Queens Problem` in our classes.

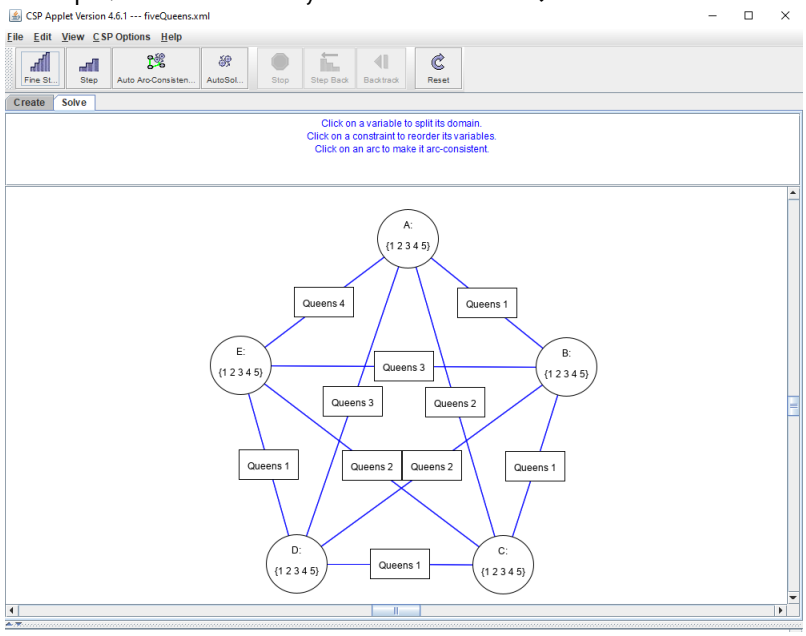

Figure 3.1. Five Queens in `constriant.jar` Applet

There are two tabs in the applet, namely `Create` and `Solve`. In the `Create` tab, you can create new CSPs or edit an existing one. It allows you to create variables, create constraints, connect variables to constraints, select and move

the placed objects, delete objects, and set properties. In the `Solve` tab, you can assign values for variables, apply arc consistency, apply backtracking to solve problems, etc.
Play around with the applet to get an idea about how it works. Then formulate and solve the following tasks using it.

2 Question 1 (5 points): Eating Out

Zahid (Z), Ishrak (I), Farabi (F), and Nafisa (N) came to Chini-Come, a restaurant near their university. The restaurant serves Special Rice (S), Biriyani Rice (B), Kashmiri Naan (K), and Paratha (P). You overhear their conversations, and come up with the following preferences:

- Zahid does not like Paratha.
- Ishrak and Farabi want to grab a bite of each other's food. So they want to order different dishes.
- Farabi likes Rice items. So he will either take Special Rice or Biriyani Rice.
- Zahid wants to take a unique dish. However, he loves to copy Ishrak and will order the same dish as Ishrak.
- Nafisa will not order Kashmiri Naan as she had them earlier.

Formulate the problem as CSP and explore the possible solution(s).

3 Question 2 (6 points): Finding Houses

Four people, Ali (A), Sristy (S), Maliha (M), and Rafid (R) are looking to rent space in an apartment building. There are three floors in the building: 1, 2, and 3 (where 1 is the lowest and 3 is the highest). More than one person can live on a single floor, but each person must be assigned to some floor. The following constraints must be satisfied on the assignment:

- Ali and Sristy must not live on the same floor.
- If Ali and Maliha live on the same floor, they must both be living on floor 2.
- If Ali and Maliha live on different floors, one of them must be living on floor 3.
- Rafid must not live on the same floor as anyone else.
- Rafid must live on a higher floor than Maliha.

Formulate the problem as CSP and explore the possible solution(s).

4 Question 3 (7 points): Spots

Assume that six friends Rifat (R), Atiq (A), Farhan (F), Ishmam (I), Tabassum (T), and Sabrina (S) are standing in a queue, each occupying a unique spot among the six possible spots labeled 1 to 6. Farhan is standing in between Atiq and Ishmam. Sabrina and Rifat are standing next to each other. Tabassum is either at the front of the line or the back of the line. Sabrina has one person behind her.
Formulate the problem as CSP and explore the possible solution(s).

5 Question 4 (10 points): Scheduling Tasks

You need to prepare a schedule for two faculty members, X and Y. They need to carry out the following tasks:

(G) Gather contents for Database Management Systems (DBMS) Lab, which takes 1 hour.

(Q) Check quiz scripts, which takes 2 consecutive hours.

(C) Take Artificial Intelligence (AI) class, which takes 1 hour.

(D) Conduct DBMS Lab, which takes 1 hour.

(L) Take AI lab, which takes 2 consecutive hours.

The schedule consists of one-hour slots: 8 am - 9 am, 9 am - 10 am, 10 am - 11 am, 11 am - 12 pm. The requirements are as follows:

- At any given time, each faculty member can do at most one task (G, Q, C, D, L).
- The AI class (C) must happen before AI lab (L).
- The contents (G) should be gathered before taking the DBMS Lab (D).
- The DBMS Lab (D) should be finished by 10 am.
- X is going to gather contents for DBMS (G) since s/he's good at browsing contents.
- The other faculty member not conducting DBMS lab (D) should attend the lab, and hence cannot do anything else at that time.
- The person taking DBMS Lab (D) does not take AI Lab (L)
- The person taking AI Lab (L) must also take the AI class (C)
- Checking quiz scripts (Q) takes 2 consecutive hours and hence should start at or before 10 am.
- Taking AI Lab (L) takes 2 consecutive hours and hence should start at or before 10 am.

Formulate the problem as CSP and explore the possible solution(s).

6 Evaluation

Once you are done with the tasks, call your course teacher and show them your implementations.

7 Submission

Submit one file: `StudentID_L3.pdf` (StudentID will be replaced by your student ID) under **Assignment 3** on **Google Classroom**. The report can contain (but is not limited to) **introduction** (briefly introduce the problem you tackled, highlight the specific algorithm/technique you implemented), **problem analysis** (discuss the nuances of the problem, explain what modification or unique aspects you were asked to introduce), **solution explanation** (provide an overview of your approach, include screenshots for clarification, highlight key decisions you made during the implementation), **findings and insights** (share any interesting observations/insights you gained, discuss how your solution performed in various scenarios), **challenges faced** (describe any challenges you encountered, explain how you overcame these challenges), **hyperparameter exploration** (discuss the impact of different hyperparameter values on your solution's behavior, analyze how changes influenced the performance), **additional information** (feel free to include any extra details, such as extensions, additional features, or creative elements you added to your solution). The report is about insights, analysis, and reflection — no need to duplicate your entire code. Be clear and concise in your explanations. This is your chance to showcase your understanding and creativity, so make the most of it.
You will have **2 weeks** to submit the file. Plagiarism is strictly prohibited and will result in significant penalties.

Lab 4 Multi-Agent Search

In this lab, you will design agents for the classic version of Pacman, including ghosts. Along the way, you will implement both minimax and expectimax search and try your hand at evaluation function design.
The code base has not changed much from the previous lab, but please start with a fresh installation, rather than intermingling files from lab 1.
As in lab 0, this lab includes an autograder for you to grade your answers on your machine. This can be run on all questions with the command:

```
python autograder.py
```

It can be run for one particular question, such as q2, by:

```
python autograder.py -q q2
```

It can be run for one particular test by commands of the form:

```
python autograder.py -t test_cases/q2/0-small-tree
```

By default, the autograder displays graphics with the `-t` option but does not with the `-q` option. You can force graphics by using the `--graphics` flag, or force no graphics by using the `--no-graphics` flag.
See the autograder tutorial in Lab 0 for more information about using the autograder.
The code for this lab contains the following files, available as `multiagent.zip`.

Files you will edit:	
`multiAgents.py`	Where all of your multi-agent search agents will reside.
Files you might want to look at:	
`pacman.py`	The main file that runs Pacman games. This file describes a Pacman GameState type, which you use in this lab.
`game.py`	The logic behind how the Pacman world works. This file describes several supporting types like AgentState, Agetn Direction, and Grid
`util.py`	Useful data structures for implementing search algorithms.
Supporting files you can ignore:	
`graphicsDisplay.py`	Graphics for Pacman
`graphicsUtils.py`	Support for Pacman graphics
`textDisplay.py`	ASCII graphics for Pacman
`ghostAgents.py`	Agents to control ghosts
`keyboardAgents.py`	Keyboard interfaces to control Pacman
`layout.py`	Code for reading layout files and storing their contents
`autograder.py`	Lab autograder
`testParser.py`	Parses autograder test and solution files
`testClasses.py`	General autograding test classes
`test_cases/`	Directory containing the test cases for each question
`multiagentTestClasses.py`	Task 2 specific autograding test classes

Files to Edit and Submit: You will fill in portions of `multiAgents.py` during the task. Please *do not* change the other files in this distribution or submit any of our original files other than this file.

Evaluation: Your code will be autograded for technical correctness. Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation – not the autograder's judgments – will be the final judge of your score. We will review and grade assignments individually to ensure that you receive due credit for your work.
Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please do not try. We trust you all to submit your own work only; please do not let us down. If you do, we will pursue the strongest consequences available to us.
Getting Help: You are not alone! If you find yourself stuck on something, contact us for help. Office hours, Google Classroom, and Emails are there for your support; please use them. We want these labs to be rewarding and instructional, not frustrating and demoralizing. But, we do not know when or how to help unless you ask.
Google Classroom: Please be careful not to post spoilers.
Report Due: 2 weeks after the lab. Please check the submission deadline in the post for more details.

1 Welcome to Multi-Agent Pacman

First, play a game of classic Pacman by running the following command:

```
python pacman.py
```

and using the arrow keys to move. Now, run the provided `ReflexAgent` in `multiAgents.py`

```
python pacman.py -p ReflexAgent
```

Note that it plays quite poorly even on simple layouts:

```
python pacman.py -p ReflexAgent -l testClassic
```

Inspect its code (in `multiAgents.py`) and make sure you understand what it is doing.

2 Question 1 (4 points): Reflex Agent

Improve the `ReflexAgent` in `multiAgents.py` to play respectably. The provided reflex agent code provides some helpful examples of methods that query the `GameState` for information. A capable reflex agent will have to consider both food locations and ghost locations to perform well. Your agent should easily and reliably clear the `testClassic` layout:

```
python pacman.py -p ReflexAgent -l testClassic
```

Try out your reflex agent on the default `mediumClassic` layout with one ghost or two (and animation off to speed up the display):

```
python pacman.py --frameTime 0 -p ReflexAgent -k 1
python pacman.py --frameTime 0 -p ReflexAgent -k 2
```

How does your agent fare? It will likely often die with 2 ghosts on the default board unless your evaluation function is quite good.
`Note:` Remember that `newFood` has the function `asList()`
Note: As features, try the reciprocal of important values (such as distance to food) rather than just the values themselves.
Note: The evaluation function you are writing is evaluating state-action pairs; in later parts of the project, you will be evaluating states.
Note: You may find it useful to view the internal contents of various objects for debugging. You can do this by printing the objects' string representations. For example, you can print `newGhostStates` with `print(str(newGhostStates))`.
Options: Default ghosts are random; you can also play for fun with slightly smarter directional ghosts using `-g DirectionalGhost`. If the randomness is preventing you from telling whether your agent is improving, you can

use `-f` to run with a fixed random seed (same random choices every game). You can also play multiple games in a row with `-n`. Turn off graphics with `-q` to run lots of games quickly.

Grading: We will run your agent on the `openClassic` layout 10 times. You will receive 0 points if your agent times out, or never wins. You will receive 1 point if your agent wins at least 5 times, or 2 points if your agent wins all 10 games. You will receive an additional 1 point if your agent's average score is greater than 500, or 2 points if it is greater than 1000. You can try your agent out under these conditions with

```
python autograder.py -q q1
```

To run it without graphics, use:

```
python autograder.py -q q1 --no-graphics
```

Do not spend too much time on this question, though, as the meat of the lab lies ahead.

3 Question 2 (5 points): Minimax

Now you will write an adversarial search agent in the provided `MinimaxAgent` class stub in `multiAgents.py`. Your minimax agent should work with any number of ghosts, so you will have to write an algorithm that is slightly more general than what you have previously seen in the lecture. In particular, your minimax tree will have multiple min layers (one for each ghost) for every max layer.

Your code should also expand the game tree to an arbitrary depth. Score the leaves of your minimax tree with the supplied `self.evaluationFunction`, which defaults to `scoreEvaluationFunction`. `MinimaxAgent` extends `MultiAgentSearchAgent`, which gives access to `self.depth` and `self.evaluationFunction`. Make sure your minimax code makes reference to these two variables where appropriate as these variables are populated in response to command line options.

Important: A single search ply is considered to be one Pacman move and all the ghosts' responses, so a depth 2 search will involve Pacman and each ghost moving two times (see Figure 4.1).

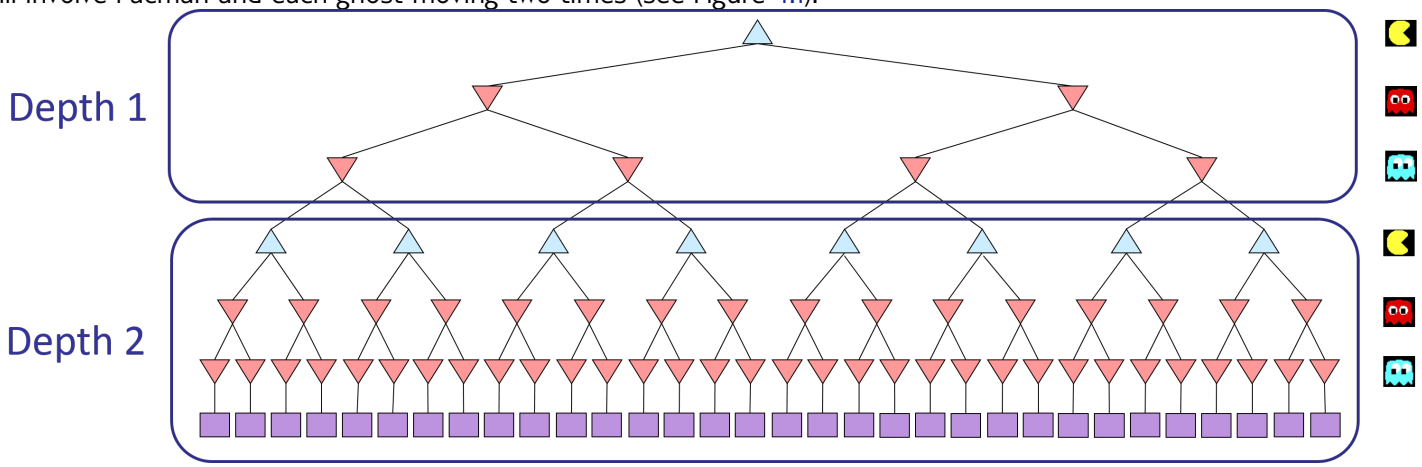

Figure 4.1. Explanation of ply

Grading: We will be checking your code to determine whether it explores the correct number of game states. This is the only reliable way to detect some very subtle bugs in implementations of minimax. As a result, the autograder will be very picky about how many times you call `GameState.generateSuccessor`. If you call it any more or less than necessary, the autograder will complain. To test and debug your code, run

```
python autograder.py -q q2
```

This will show what your algorithm does on a number of small trees, as well as a Pacman game. To run it without graphics, use:

```
python autograder.py -q q2 --no-graphics
```

? Hints and Observations

- Implement the algorithm recursively using helper function(s).
- The correct implementation of minimax will lead to Pacman losing the game in some tests. This is not a problem: as it is correct behavior, it will pass the tests.
- The evaluation function for the Pacman test in this part is already written (`self.evaluation Function`). You should not change this function but recognize that now we are evaluating states rather than actions, as we were for the reflex agent. Look-ahead agents evaluate future states whereas reflex agents evaluate actions from the current state.
- The minimax values of the initial state in the `minimaxClassic` layout are 9, 8, 7, -492 for depths 1, 2, 3, and 4 respectively. Note that your minimax agent will often win (665/1000 games for us) despite the dire prediction of depth 4 minimax.

```
python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
```

- Pacman is always agent 0, and the agents move in order of increasing agent index.
- All states in minimax should be `GameStates`, either passed in to `getAction` or generated via `GameState.generateSuccessor`. In this lab, you will not be abstracting to simplified states.
- On larger boards such as `openClassic` and `mediumClassic` (the default), you will find Pacman to be good at not dying, but quite bad at winning. He will often thrash around without making progress. He might even thrash around right next to a dot without eating it because he doesn't know where he would go after eating that dot. Do not worry if you see this behavior, question 5 will clean up all of these issues.
- When Pacman believes that his death is unavoidable, he will try to end the game as soon as possible because of the constant penalty for living. Sometimes, this is the wrong thing to do with random ghosts, but minimax agents always assume the worst:

```
python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3
```

Make sure you understand why Pacman rushes the closest ghost in this case.

4 Question 3 (5 points): Alpha-Beta Pruning

Make a new agent that uses alpha-beta pruning to efficiently explore the minimax tree, in `AlphaBetaAgent` from `multiagents.py`. Your algorithm will be slightly more general than the pseudocode from the lecture, so part of the challenge is to extend the alpha-beta pruning logic appropriately to multiple minimizer agents.
You should see a speed-up (perhaps depth 3 alpha-beta will run as fast as depth 2 minimax). Ideally, depth 3 on `smallClassic` should run in just a few seconds per move or faster.

```
python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
```

The `AlphaBetaAgent` minimax values should be identical to the `MinimaxAgent` minimax values, although the actions it selects can vary because of different tie-breaking behavior. Again, the minimax values of the initial state in the `minimaxClassic` layout are 9, 8, 7, and -492 for depths 1, 2, 3, and 4 respectively.
Grading: Because we check your code to determine whether it explores the correct number of states, it is important that you perform alpha-beta pruning without reordering children. In other words, successor states should always be processed in the order returned by `GameState.getLegalActions`. Again, do not call `GameState.generateSuccessor` more than necessary.
You must not prune on equality in order to match the set of states explored by our autograder. (Indeed, alternatively, but incompatible with our autograder, would be to also allow for pruning on equality and invoke alpha-beta once on each child of the root node, but this will not match the autograder.)
The pseudo-code below represents the algorithm you should implement for this question:

α: MAX's best option on path to root
β: MIN's best option on path to root

def max-value(state, α, β)
initialize $v = -\infty$
for each successor of state:
$v = \max(v,$ value(successor, $\alpha, \beta)$)
if $v > \beta$ return v
$\alpha = \max(\alpha, v)$
return v

def min-value(state, α, β)
initialize $v = +\infty$
for each successor of state:
$v = \min(v,$ value(successor, $\alpha, \beta)$)
if $v < \alpha$ return v
$\beta = \min(\beta, v)$
return v

To test and debug your code, run

```
python autograder.py -q q3
```

This will show what your algorithm does on a number of small trees, as well as a Pacman game. To run it without graphics, use:

```
python autograder.py -q q3 --no-graphics
```

The correct implementation of alpha-beta pruning will lead to Pacman losing some of the tests. This is not a problem: as it is correct behavior, it will pass the tests.

5 Question 4 (5 points): Expectimax

Minimax and alpha-beta are great, but they both assume that you are playing against an adversary who makes optimal decisions. As anyone who has ever won tic-tac-toe can tell you, this is not always the case. In this question, you will implement the `ExpectimaxAgent`, which is useful for modeling the probabilistic behavior of agents who may make suboptimal choices. The pseudo-code below represents the algorithm you should implement for this question:

def exp-value(state):
initialize $v = 0$
for each successor of state:
$p = probability($successor$)$
$v += p \times value($successor$)$
return v

As with the search and constraint satisfaction problems covered so far in this class, the beauty of these algorithms is their general applicability. To expedite your own development, we have supplied some test cases based on generic trees. You can debug your implementation on small the game trees using the command:

```
python autograder.py -q q4
```

Debugging on these small and manageable test cases is recommended and will help you to find bugs quickly.
Once your algorithm is working on small trees, you can observe its success in Pacman. Random ghosts are of course not optimal minimax agents, so modeling them with minimax search may not be appropriate. `ExpectimaxAgent`, will no longer take the min over all ghost actions, but the expectation according to your agent's model of how the ghosts act. To simplify your code, assume you will only be running against an adversary which chooses amongst their `getLegalActions` uniformly at random while you try to play optimally.
To see how the `ExpectimaxAgent` behaves in Pacman, run:

```
python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3
```

You should now observe a more cavalier approach in close quarters with ghosts. In particular, if Pacman perceives that he could be trapped but might escape to grab a few more pieces of food, he will at least try. Investigate the results of these two scenarios:

```
python pacman.py -p AlphaBetaAgent -l trappedClassic -a depth=3 -q -n 10
python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10
```

You should find that your `ExpectimaxAgent` wins about half the time, while your `AlphaBetaAgent` always loses. Make sure you understand why the behavior here differs from the minimax case.
The correct implementation of expectimax will lead to Pacman losing some of the tests. This is not a problem: as it is correct behavior, it will pass the tests.

6 Question 5 (6 points): Evaluation Function

Write a better evaluation function for Pacman in the provided function `betterEvaluationFunction` from `multiAgents.py`. The evaluation function should evaluate states, rather than actions like your reflex agent evaluation function did in Question 1. You may use any tools at your disposal for evaluation, including your search code from the last project. With depth 2 search, your evaluation function should clear the `smallClassic` layout with one random ghost more than half the time and still run at a reasonable rate (to get full credit, Pacman should be averaging around 1000 points when he's winning).
Grading: the autograder will run your agent on the `smallClassic` layout 10 times. We will assign points to your evaluation function in the following way:

- If you win at least once without timing out the autograder, you receive 1 point. Any agent not satisfying these criteria will receive 0 points.
- +1 for winning at least 5 times, +2 for winning all 10 times
- +1 for an average score of at least 500, +2 for an average score of at least 1000 (including scores on lost games)
- +1 if your games take on average less than 30 seconds on the autograder machine when run with `-no-graphics`. The autograder is run on a specialized lab PC/server, so this machine will have a fair amount of resources, but your personal computer could be far less performant (netbooks) or far more performant (gaming rigs).
- The additional points for average score and computation time will only be awarded if you win at least 5 times.

You can try your agent out under these conditions with

```
python autograder.py -q q5
```

To run it without graphics, use:

```
python autograder.py -q q5 --no-graphics
```

7 Evaluation

Once you are done with the tasks, call your course teacher and show them the autograder results and codes.

8 Submission

Submit one file: `StudentID_L4.pdf` (`StudentID` will be replaced by your student ID) under **Assignment 4** on **Google Classroom**. The report can contain (but is not limited to) **introduction** (briefly introduce the problem you tackled, highlight the specific algorithm/technique you implemented), **problem analysis** (discuss the nuances of the problem, explain what modification or unique aspects you were asked to introduce), **solution explanation** (provide an overview of your approach, include portions of the code only where necessary for clarification, highlight key decisions

you made during the implementation, analyze the complexity of your code), **findings and insights** (share any interesting observations/insights you gained, discuss how your solution performed in various scenarios), **challenges faced** (describe any challenges you encountered, explain how you overcame these challenges), **hyperparameter exploration** (discuss the impact of different hyperparameter values on your solution's behavior, analyze how changes influenced the performance), **additional information** (feel free to include any extra details, such as extensions, additional features, or creative elements you added to your solution). The report is about insights, analysis, and reflection — no need to duplicate your entire code. Be clear and concise in your explanations. This is your chance to showcase your understanding and creativity, so make the most of it.

You will have **2 weeks** to submit the file. Plagiarism is strictly prohibited and will result in significant penalties.

Lab 5 Markov Decision Process

In this lab, you will implement value iteration. You will test your agent on Gridworld (from class).
As in previous tasks, this lab includes an autograder for you to grade your answers on your machine. This can be run on all questions with the command:

```
python autograder.py
```

It can be run for one particular question, such as q2, by:

```
python autograder.py -q q2
```

It can be run for one particular test by commands of the form:

```
python autograder.py -t test_cases/q2/1-bridge-grid
```

The code for this lab contains the following files, available as `mdp.zip`.

Files you will edit:	
`valueIterationAgents.py`	A value iteration agent for solving known MDPs.
`analysis.py`	A file to put your answers to questions given in the lab.
Files you might want to look at:	
`mdp.py`	Defines methods on general MDPs.
`learningAgents.py`	Defines the base classes `ValueEstimationAgent`, which your agents will extend.
`util.py`	Useful data structures for implementing MDPs.
`gridworld.py`	The Gridworld implementation.
Supporting files you can ignore:	
`environment.py`	Abstract class for general reinforcement learning environments. Used by `gridworld.py`.
`graphicsGridworldDisplay.py`	Gridworld graphical display.
`graphicsUtils.py`	Graphics utilities.
`textGridworldDisplay.py`	Plug-in for the Gridworld text interface.
`crawler.py`	The crawler code and test harness. You will run this but not edit it.
`graphicsCrawlerDisplay.py`	GUI for the crawler robot.
`qlearningAgents.py`	Q-learning agents for Gridworld, Crawler and Pacman.
`featureExtractors.py`	Classes for extracting features on (state, action) pairs. Used for the approximate Q-learning agent (in `qlearningAgents.py`).
`autograder.py`	Lab autograder
`testParser.py`	Parses autograder test and solution files
`testClasses.py`	General autograding test classes
`test_cases/`	Directory containing the test cases for each question
`reinforcementTestClasses.py`	Lab specific autograding test classes

Files to Edit: You will fill in portions of `valueIterationAgents.py` and `analysis.py` during the task. Please *do not* change the other files in this distribution or submit any of our original files other than this file.

Evaluation: Your code will be autograded for technical correctness. Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation – not the autograder's judgments – will be the final judge of your score. We will review and grade assignments individually to ensure that you receive due credit for your work.
Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please do not try. We trust you all to submit your own work only; please do not let us down. If you do, we will pursue the strongest consequences available to us.
Getting Help: You are not alone! If you find yourself stuck on something, contact us for help. Office hours, Google Classroom, and Emails are there for your support; please use them. We want these labs to be rewarding and instructional, not frustrating and demoralizing. But, we do not know when or how to help unless you ask.
Google Classroom: Please be careful not to post spoilers.
Report Due: 2 weeks after the lab. Please check the submission deadline in the post for more details.

1 MDPs

To get started, run Gridworld in manual control mode, which uses the arrow keys:

```
python gridworld.py -m
```

You will see the two-exit layout from class. The blue dot is the agent. Note that when you press up, the agent only actually moves north 80% of the time. Such is the life of a Gridworld agent!
You can control many aspects of the simulation. A full list of options is available by running:

```
python gridworld.py -h
```

The default agent moves randomly

```
python gridworld.py -g MazeGrid
```

You should see the random agent bounce around the grid until it happens upon an exit. Not the finest hour for an AI agent.
Note: The Gridworld MDP is such that you first must enter a pre-terminal state (the double boxes shown in the GUI) and then take the special 'exit' action before the episode ends (in the true terminal state called `TERMINAL_STATE`, which is not shown in the GUI). If you run an episode manually, your total return may be less than you expected, due to the discount rate (`-d` to change; 0.9 by default).
Look at the console output that accompanies the graphical output (or use `-t` for all text). You will be told about each transition the agent experiences (to turn this off, use `-q`).
As in Pacman, positions are represented by `(x, y)` Cartesian coordinates, and any arrays are indexed by `[x][y]`, with '`north`' being the direction of increasing `y`, etc. By default, most transitions will receive a reward of zero, though you can change this with the living reward option (`-r`).

2 Question 1 (4 points): Value Iteration

Recall the value iteration state update equation:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V_k(s')\right]$$

Write a value iteration agent in `ValueIterationAgent`, which has been partially specified for you in `valueIterationAgents.py`. Your value iteration agent is an offline planner that takes the number of iterations (option `-i`) as input in its initial planning phase. `ValueIterationAgent` takes an MDP on construction and runs value iteration for the specified number of iterations before the constructor returns.

Value iteration computers k-step estimates of the optimal values, V_k. In addition to running value iteration, implement the following methods for `ValueIterationAgent` using V_k.

- `computeActionFromValues(state)` computes the best action according to the value function given by `self.values`.
- `computeQValueFromValues(state, action)` returns the Q-value of the (state, action) pair given by the value function given by `self.values`.

These quantities are all displayed in the GUI: values are numbers in squares, Q-values are numbers in square quarters, and policies are arrows out from each square.

Important: Use the "batch" version of value iteration where each vector V_k s computed from a fixed vector V_{k-1} (as shown in the lecture), not the "online" version where one single weight vector is updated in place. This means that when a state's value is updated in iteration k based on the values of its successor states, the successor state values used in the value update computation should be those from iteration $k-1$ (even if some of the successor states had already been updated in iteration k). The difference is discussed in Sutton & Barto in Chapter 4.1 on page 91.

Note: A policy synthesized from values of depth k (which reflect the next k rewards) will actually reflect the next $k+1$ rewards (i.e. you return π_{k+1}). Similarly, the Q-values will also reflect one more reward than the values (i.e. you return Q_{k+1}).

You should return the synthesized policy π_{k+1}.

Hint: You may optionally use the `util.Counter` class in `util.py`, which is a dictionary with a default value of zero. However, be careful with `argMax`: the actual argmax you want may be a key, not in the counter!

Note: Make sure to handle the case when a state has no available actions in an MDP (think about what this means for future rewards).

To test your implementation, run the autograder:

```
python autograder.py -q q1
```

The following command loads your `ValueIterationAgent`, which will compute a policy and execute it 10 times. Press a key to cycle through values, Q-values, and the simulation. You should find that the value of the start state (`V(start)`, which you can read off of the GUI) and the empirical resulting average reward (printer after the 10 rounds of execution finish) are quite close.

```
python gridworld.py -a value -i 100 -k 10
```

Hint: On the default BookGrid, you can run value iteration for 5 iterations using the following command:

```
python gridworld.py -a value -i 5
```

The result of running the code should look like Figure 5.1.

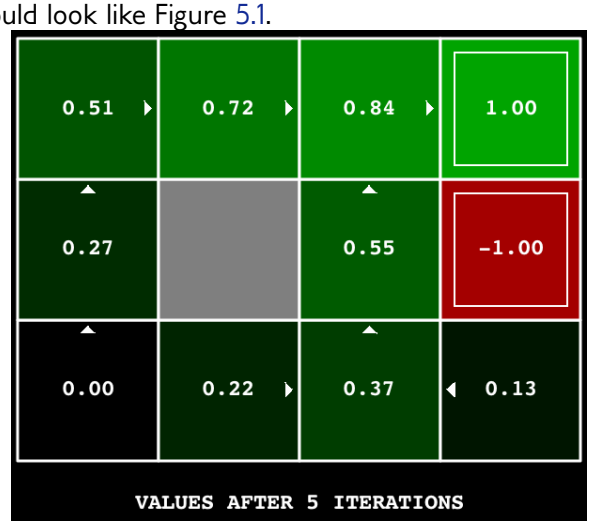

Figure 5.1. Result of running value iteration for 5 iterations in BookGrid

Grading: Your value iteration agent will be graded in a new grid. We will check your values, Q-values, and policies after fixed numbers of iterations and at convergence (e.g. after 100 iterations).

3 Question 2 (1 point): Bridge Crossing Analysis

`BridgeGrid` is a grid world map with a low-reward terminal state and a high-reward terminal state separated by a narrow "bridge", on either side of which is a chasm of high negative reward. Here, the expected value of a state depends on a number of factors, including how the future rewards are discounted, how noisy the actions are, and how much reward is received in the non-terminal states. The agent starts near the low-reward state. With the default discount of 0.9 and the default noise of 0.2, the optimal policy does not cross the bridge. Here, noise refers to how often an agent ends up in an unintended successor state when they perform an action. And discount determines how much the agent cares about rewards in the distant future relative to those in the immediate future. Change only ONE of the discount and noise parameters so that the optimal policy causes the agent to attempt to cross the bridge. Put your answer in `question2()` of `analysis.py`. The default can be seen using the following command:

```
python gridworld.py -a value -i 100 -g BridgeGrid --discount 0.9 --noise 0.2
```

The resultant grid can be seen in Figure 5.2.

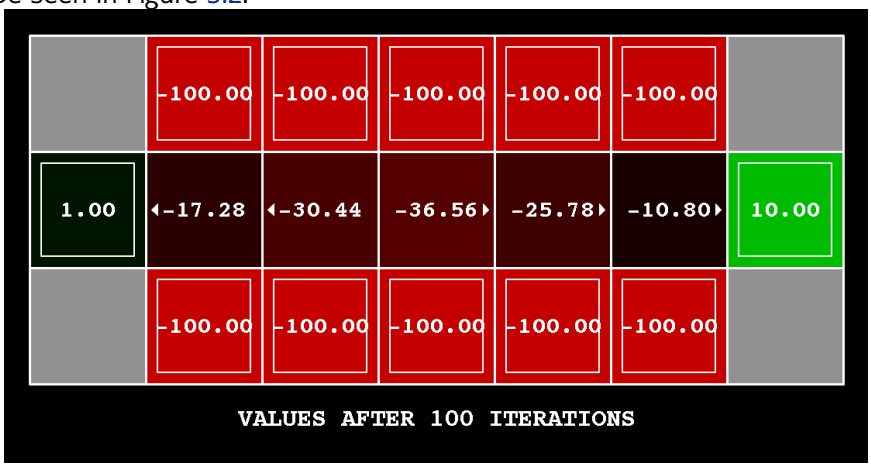

Figure 5.2. Default Values for Question 2

Grading: We will check that you only changed one of the given parameters and that with this change, a correct value iteration agent should cross the bridge. To check your answer, run the autograder:

```
python autograder.py -q q2
```

4 Question 3 (5 points): Policies

Consider the `DiscountGrid` layout, shown in Figure 5.3. This grid has two terminal states with a positive payoff (in the middle row) – a close (near) exit with payoff +1, and a distant exit with payoff +10. The bottom row of the grid consists of terminal states with negative payoff (shown in red); each state in this "cliff" region has a payoff of -10. The starting state is the yellow square. We distinguish between two types of paths:

1. paths that "risk the cliff" and travel near the bottom row of the grid; these paths are shorter but risk earning a large negative payoff, and are represented by the red arrow in Figure 5.3.
2. paths that "avoid the cliff" and travel along the top edge of the grid. These paths are longer but are less likely to incur huge negative payoffs. These paths are represented by the green arrow in Figure 5.3.

In this question, you will choose values of the discount, noise, and living reward parameters for this MDP to produce optimal policies of several different types that are given below. **Your setting of the parameter values for each part should have the property that, if your agent followed its optimal policy without being subject to any noise, it would exhibit the given behavior.** If a particular behavior is not achieved for any setting of the parameters, assert that the policy is impossible by returning the string '`NOT POSSIBLE`'.

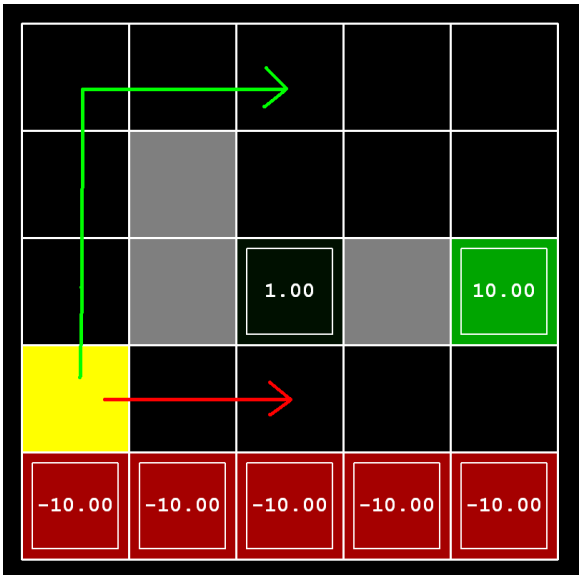

Figure 5.3. DiscountGrid Layout

Here are the optimal policy types you should attempt to produce:

a. Prefer the close exit (+1), risking the cliff (-10)

b. Prefer the close exit (+1), avoiding the cliff (-1)

c. Prefer the distant exit (+10), risking the cliff (-10)

d. Prefer the distant exit (+10), avoiding the cliff (-10)

e. Avoid both exits and the cliff (so an episode should never terminate)

To see what behavior a set of numbers ends up in, run the following command to see a GUI:

```
python gridworld.py -g DiscountGrid -a value --discount [YOUR_DISCOUNT] --noise [
    YOUR_NOISE] --livingReward [YOUR_LIVING_REWARD]
```

Write your answers in `analysis.py`: `question3a()` through `question3e()` should each return a 3-item tuple of discount, noise, and living reward.

To check your answers, run the autograder:

```
python autograder.py -q q3
```

Note: You can check your policies in the GUI with commands like this:

```
python gridworld.py -a value -i 100 -g DiscountGrid -d 0.0 -n 0.0 -r 0.0
```

This will run the value iteration agent for `100` iterations on the `DiscountGrid` layout. The discount, noise, and living reward all are set to `0`.

As shown in Figure 5.4, using a correct answer to 3(a), the arrow in (0, 1) should point east, the arrow in (1, 1) should also point east, and the arrow in (2, 1) should point north.

Note: On some machines you may not see an arrow. In this case, press a button on the keyboard to switch to the qValues display, and mentally calculate the policy by taking the argmax of the available qValues for each state.

Grading: We will check that the desired policy is returned in each case.

5 Question 4 (1 point): Asynchronous Value Iteration

Write a value iteration agent in `AsynchronousValueIterationAgent`, which has been partially specified for you in `valueIterationAgents.py`. Your value iteration agent is an offline planner that takes the number of iterations (option `-i`) as input in its initial planning phase.

`AsynchronousValueIterationAgent` takes an MDP on construction and runs *cyclic* value iteration (described

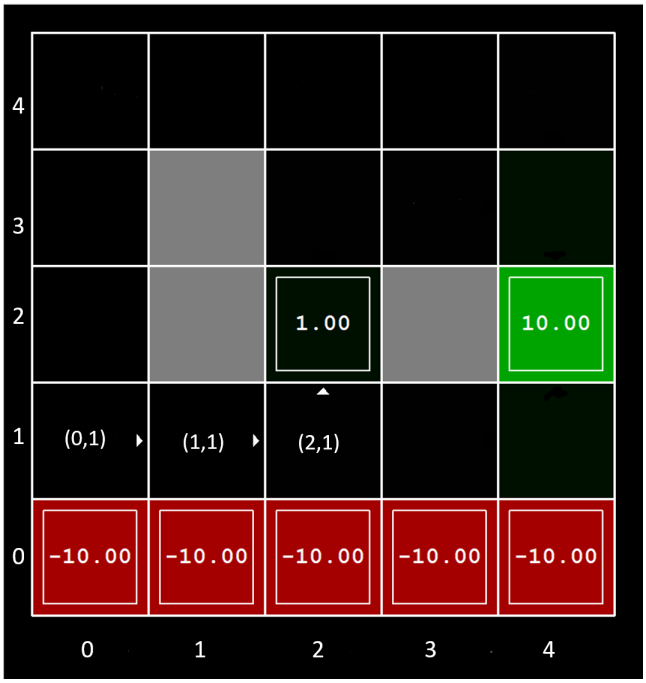

Figure 5.4. Sample policy with the index shown in Parentheses

in the next paragraph) for the specified number of iterations before the constructor returns. Note that all this value iteration code should be placed inside the constructor (`__init__` method).

The reason this class is called `AsynchornousValueIterationAgent` is because we will update only **one** state in each iteration, as opposed to doing a batch-style update. Here is how cyclic value iteration works. In the first iteration, only update the value of the first state in the state's list. In the second iteration, only update the value of the second. Keep going until you have updated the value of each state once, then start back at the first state for the subsequent iteration. **If the state picked for updating is terminal, nothing happens in that iteration.** You can implement it as indexing into the states variable defined in the code skeleton.

As a reminder, here's the value iteration state update equation:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V_k(s')\right]$$

Value iteration iterates a fixed-point equation, as discussed in class. It is also possible to update the state values in different ways, such as in a random order (i.e., select a state randomly, update its value, and repeat) or in a batch style (as in Q1). In Q4, we will explore another technique. We want to iterate through the states in the order provided by the `getStates()` functions and update one state per iteration.

`AsynchronousValueIterationAgent` inherits from `ValueIterationAgent` from Q1, so the only method you need to implement is `runValueIteration`. Since the superclass constructor calls `runValueIteration`, overriding it is sufficient to change the agent's behavior as desired.

Note: Make sure to handle the case when a state has no available actions in an MDP (think about what this means for future rewards).

To test your implementation, run the autograder. It should take less than a second to run.

```
python autograder.py -q q4
```

The following command loads your `AsynchronousValueIterationAgent` in the Gridworld, which will compute a policy and execute it 10 times. Press a key to cycle through values, Q-values, and the simulation. You should find that the value of the start state (`V(start)`), which you can read off of the GUI) and the empirical resulting average reward (printed after the 10 rounds of execution finish) are quite close.

```
python gridworld.py -a asynchvalue -i 1000 -k 10
```

Grading: Your value iteration agent will be graded on a new grid. We will check your values, Q-values, and policies after fixed numbers of iterations and at convergence (e.g., after 1000 iterations).

6 Question 5 (1 point): Prioritized Sweeping Value Iteration

You will now implement `PrioritizedSweepingValueIterationAgent`, which has been partially specified for you in `valueIterationAgents.py`.
Note that this class derives from `AsynchronousValueIterationAgent`, so the only method that needs to change is `runValueIteration`, which actually runs the value iteration.
Prioritized sweeping attempts to focus updates of state values in ways that are likely to change the policy.
For this lab, you will implement a simplified version of the standard prioritized sweeping algorithm, which is described in this paper. We have adapted this algorithm for our setting. First, we define the **predecessors** of a state `s` as all states that have a **nonzero** probability of reaching `s` by taking some action `a`. Also, `theta`, which is passed in as a parameter, will represent our tolerance for error when deciding whether to update the value of a state. Here's the algorithm you should follow in your implementation:

- Computer predecessors of all states. To do that, get all states, iterated over them to check if it is a terminal state or not. If it is not a terminal state, find all successor in which can end up from this state using all the available actions.
- Initialize an empty priority queue.
- For each non-terminal state `s`, do: (note: to make the autograder work for this question, you must iterate over states in the order returned by `self.mdp.getStates()`).
  - Find the absolute value of the difference between the current value of `s` in `self.values` and the highest Q-value across all possible actions from `s` (this represents what the value should be); call this number `diff`. Do NOT update `self.values[s]` in this step.
  - Push `s` into the priority queue with priority `-diff` (note that this is **negative**). We use a negative because the priority queue is a min-heap, but we want to prioritize updating states that have a **higher** error.
- For `iteration` in `0, 1, 2, ..., self.iterations - 1`, do:
  - If the priority queue is empty, then terminate.
  - Pop a state `s` off the priority queue.
  - Update `s`'s value (if it is not a terminal state) in `self.values`.
  - For each predecessor `p` of `s`, do:
    - Find the absolute value of the difference between the current value of `p` in `self.values` and the highest Q-value across all possible actions from `p` (this represents what the value should be); call this number `diff`. Do NOT update `self.values[p]` in this step.
    - If `diff > theta`, push `p` into the priority queue with priority `-diff` (note that this is **negative**), as long as it does not already exist in the priority queue with equal or lower priority. As before, we use a negative because the priority queue is a min-heap, but we want to prioritize updating states that have a **higher** error.

A couple of important notes on implementation:

- When you compute predecessors of a state, make sure to store them in a **set**, not a list, to avoid duplicates.
- Please use `util.PriorityQueue` in your implementation. The `update` method in this class will likely be useful; look at its documentation.

To test your implementation, run the autograder. It should take about 1 second to run.

```
python autograder.py -q q5
```

You can run the `PrioritizedSweepingValueIterationAgent` in the Gridworld using the following command:

```
python gridworld.py -a priosweepvalue -i 1000
```

Grading: Your prioritized sweeping value iteration agent will be graded on a new grid. We will check your values, Q-values, and policies after fixed numbers of iterations and at convergence (e.g., after 1000 iterations).

7 Evaluation

Once you are done with the tasks, call your course teacher and show them the autograder results and codes.

8 Submission

Submit one file: `StudentID_L5.pdf` (`StudentID` will be replaced by your student ID) under **Assignment 5** on **Google Classroom**. The report can contain (but is not limited to) **introduction** (briefly introduce the problem you tackled, highlight the specific algorithm/technique you implemented), **problem analysis** (discuss the nuances of the problem, explain what modification or unique aspects you were asked to introduce), **solution explanation** (provide an overview of your approach, include portions of the code only where necessary for clarification, highlight key decisions you made during the implementation, analyze the complexity of your code), **findings and insights** (share any interesting observations/insights you gained, discuss how your solution performed in various scenarios), **challenges faced** (describe any challenges you encountered, explain how you overcame these challenges), **hyperparameter exploration** (discuss the impact of different hyperparameter values on your solution's behavior, analyze how changes influenced the performance), **additional information** (feel free to include any extra details, such as extensions, additional features, or creative elements you added to your solution). The report is about insights, analysis, and reflection — no need to duplicate your entire code. Be clear and concise in your explanations. This is your chance to showcase your understanding and creativity, so make the most of it.
You will have 2 weeks to submit the file. Plagiarism is strictly prohibited and will result in significant penalties.

Lab 6 Reinforcement Learning

This is a continuation of the previous lab. In this lab, you will implement Q-learning. You will test your agent first on Gridworld (from class), then apply them to a simulated robot controller (Crawler) and Pacman.
As in previous tasks, this lab includes an autograder for you to grade your answers on your machine. This can be run on all questions with the command:

```
python autograder.py
```

It can be run for one particular question, such as q5, by:

```
python autograder.py -q q5
```

The code for this lab contains the following files, available as `reinforcement.zip`.

Files you will edit:	
`qlearningAgents.py`	Q-learning agents for Gridworld, Crawler and Pacman.
`analysis.py`	A file to put your answers to questions given in the lab.
Files you might want to look at:	
`learningAgents.py`	Defines the base classes `ValueEstimationAgent` and `QLearningAgent`, which your agents will extend.
`util.py`	Utilities, including `util.Counter`, which is particularly useful for Q-learners.
`gridworld.py`	The Gridworld implementation.
`featureExtractors.py`	Classes for extracting features on (state, action) pairs. Used for the approximate Q-learning agent (in `qlearningAgents.py`).
Supporting files you can ignore:	
`mdp.py`	Defines methods on general MDPs.
`valueIterationAgents.py`	A value iteration agent for solving known MDPs.
`environment.py`	Abstract class for general reinforcement learning environments. Used by `gridworld.py`.
`graphicsGridworldDisplay.py`	Gridworld graphical display.
`graphicsUtils.py`	Graphics utilities.
`textGridworldDisplay.py`	Plug-in for the Gridworld text interface.
`crawler.py`	The crawler code and test harness. You will run this but not edit it.
`graphicsCrawlerDisplay.py`	GUI for the crawler robot.
`autograder.py`	Lab autograder
`testParser.py`	Parses autograder test and solution files
`testClasses.py`	General autograding test classes
`test_cases/`	Directory containing the test cases for each question
`reinforcementTestClasses.py`	Lab specific autograding test classes

Files to Edit: You will fill in portions of `qlearningAgents.py` and `analysis.py` during the task. Please *do not* change the other files in this distribution or submit any of our original files other than this file.

Evaluation: Your code will be autograded for technical correctness. Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation – not the autograder's judgments – will be the final judge of your score. We will review and grade assignments individually to ensure that you receive due credit for your work.
Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please do not try. We trust you all to submit your own work only; please do not let us down. If you do, we will pursue the strongest consequences available to us.
Getting Help: You are not alone! If you find yourself stuck on something, contact us for help. Office hours, Google Classroom, and Emails are there for your support; please use them. We want these labs to be rewarding and instructional, not frustrating and demoralizing. But, we do not know when or how to help unless you ask.
Google Classroom: Please be careful not to post spoilers.
Report Due: 2 weeks after the lab. Please check the submission deadline in the post for more details.

1 Question 6 (4 points): Q-Learning

Note that your value iteration agent does not actually learn from experience. Rather, it ponders its MDP model to arrive at a complete policy before ever interacting with a real environment. When it does interact with the environment, it simply follows the precomputed policy (e.g. it becomes a reflex agent). This distinction may be subtle in a simulated environment like a Gridworld, but it's very important in the real world, where the real MDP is not available.
You will now write a Q-learning agent, which does very little on construction, but instead learns by trial and error from interactions with the environment through its `update(state, action, nextState, reward)` method. A stub of a Q-learner is specified in `QLearningAgent` in `qlearningAgents.py`, and you can select it with the option '`-a q`'. For this question, you must implement the update, `computeValueFromQValues`, `getQValue`, and `computeActionFromQValues` methods.
You can use the `util.counter` method to create an extension of the dictionary type that automatically initializes all the key values to `0`. For implementing the `computeActionFromQValues` method, you should first get the maximum Q-value and then choose among one or more actions that have that Q-value.
Note: For `computeActionFromQValues`, you should break ties randomly for better behavior. The `random.choice()` function will help. When taking action, you should consider all the legal actions which also includes the actions your agent has not seen before. In a particular state, actions that your agent *hasn't* seen before still have a Q-value, specifically a Q-value of zero, and if all of the actions that your agent has seen before have a negative Q-value, an unseen action may be optimal.
Important: Make sure that in your `computeValueFromQValues` and `computeActionFromQValues` functions, you only access Q values by calling `getQValue`. This abstraction will be useful for question 10 when you override `getQValue` to use features of state-action pairs rather than state-action pairs directly.
With the Q-learning update in place, you can watch your Q-learner learn under manual control, using the keyboard:

```
python gridworld.py -a q -k 5 -m
```

Recall that `-k` will control the number of episodes your agent gets to learn. Watch how the agent learns about the state it was just in, not the one it moves to, and "leaves learning in its wake." Hint: to help with debugging, you can turn off noise by using the `-noise 0.0` parameter (though this obviously makes Q-learning less interesting). If you manually steer Pacman north and then east along the optimal path for four episodes, you should see the Q-values shown in Figure 6.1.
Grading: We will run your Q-learning agent and check that it learns the same Q-values and policy as our reference implementation when each is presented with the same set of examples. To grade your implementation, run the autograder:

```
python autograder.py -q q6
```

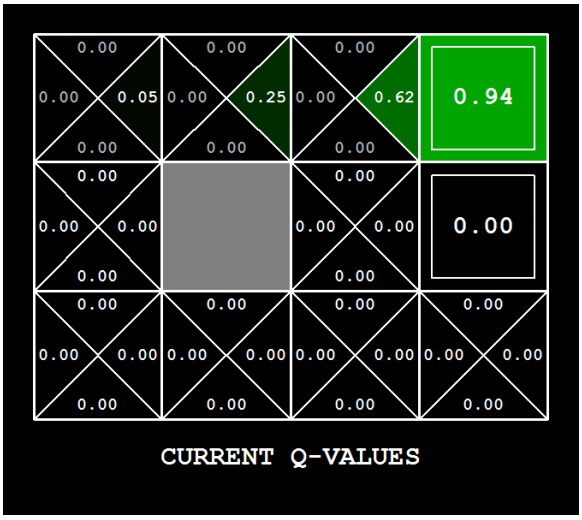

Figure 6.1. Optimal Path for Four Episodes (Manual)

2 Question 7 (2 points): Epsilon Greedy

Remember that we need an Epsilon Greedy strategy to balance exploration and exploitation. The algorithm explores ϵ% of the time and then exploits the best option k greedily. That means, initially it chooses random actions for a fraction of time and follows its current best Q-values otherwise. Now, complete your Q-learning agent by implementing epsilon-greedy action selection in `getAction`. Note that choosing a random action may result in choosing the best action - that is, you should not choose a random sub-optimal action, but rather any random legal action.

You can choose an element from a list uniformly at random by calling the `random.choice` function. You can simulate a binary variable with probability p of success by using `util.flipCoin(p)`, which returns `True` with probability p and `False` with probability $1-p$.

After implementing the `getAction` method, observe the following behavior of the agent in gridworld (with epsilon = 0.3).

```
python gridworld.py -a q -k 100
```

Your final Q-values should resemble those of your value iteration agent, especially along well-traveled paths. However, your average returns will be lower than the Q-values predict because of the random actions and the initial learning phase.

You can also observe the following simulations for different epsilon values. Does the behavior of the agent match what you expect?

```
python gridworld.py -a q -k 100 --noise 0.0 -e 0.1
python gridworld.py -a q -k 100 --noise 0.0 -e 0.9
```

To test your implementation, run the autograder:

```
python autograder.py -q q7
```

With no additional code, you should now be able to run a Q-learning crawler robot:

```
python crawler.py
```

If this does not work, you have probably written some code too specific to the `GridWorld` problem and you should make it more general to all MDPs.

This will invoke the crawling robot from the class using your Q-learner. Play around with the various learning parameters to see how they affect the agent's policies and actions. Note that the step delay is a parameter of the simulation, whereas the learning rate and epsilon are parameters of your learning algorithm, and the discount factor is a property of the environment.

3 Question 8 (1 point): Bridge Crossing Revisited

First, train a completely random Q-learner with the default learning rate on the noiseless BridgeGrid for 50 episodes and observe whether it finds the optimal policy.

```
python gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 1
```

Now try the same experiment with an epsilon of 0. Is there an epsilon and a learning rate for which it is highly likely (greater than 99%) that the optimal policy will be learned after 50 iterations? `question8()` in `analysis.py` should return EITHER a 2-item tuple of `(epsilon, learning rate)` OR the string '`NOT POSSIBLE`' if there is none. Epsilon is controlled by `-e`, learning rate by `-l`.
Note: Your response should not depend on the exact tie-breaking mechanism used to choose actions. This means your answer should be correct even if for instance we rotated the entire bridge grid world 90 degrees.
To grade your answer, run the autograder:

```
python autograder.py -q q8
```

4 Question 9 (1 point): Q-Learning and Pacman

Time to play some Pacman! Pacman will play games in two phases. In the first phase, training, Pacman will begin to learn about the values of positions and actions. Because it takes a very long time to learn accurate Q-values even for tiny grids, Pacman's training games run in quiet mode by default, with no GUI (or console) display. Once Pacman's training is complete, he will enter *testing* mode. When testing, Pacman's `self.epsilon` and `self.alpha` will be set to 0.0, effectively stopping Q-learning and disabling exploration, in order to allow Pacman to exploit his learned policy. Test games are shown in the GUI by default. Without any code changes you should be able to run Q-learning Pacman for very tiny grids as follows:

```
python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid
```

Note that `PacmanQAgent` is already defined for you in terms of the `QLearningAgent` you have already written. `PacmanQAgent` is only different in that it has default learning parameters that are more effective for the Pacman problem (`epsilon=0.05`, `alpha=0.2`, `gamma=0.8`). You will receive full credit for this question if the command above works without exceptions and your agent wins at least 80% of the time. If you have considered the unseen actions while implementing `getAction` and/or `computeActionFromQValues` methods, then you should be getting full credit for this question. The autograder will run 100 test games after the 2000 training games.
Hint: If your `QLearningAgent` works for `gridworld.py` and `crawler.py` but does not seem to be learning a good policy for Pacman on `smallGrid`, it may be because your `getAction` and/or `computeActionFromQValues` methods do not in some cases properly consider unseen actions. In particular, because unseen actions have by definition a Q-value of zero if all of the actions that have been seen have negative Q-values, an unseen action may be optimal. Beware of the argmax function from `util.Counter`!
Note: To grade your answer, run:

```
python autograder.py -q q9
```

Note: If you want to experiment with learning parameters, you can use the option `-a`, for example, `-a epsilon=0.1,alpha`
These values will then be accessible as `self.epsilon`, `self.gamma` and `self.alpha` inside the agent.
Note: While a total of 2010 games will be played, the first 2000 games will not be displayed because of the option `-x 2000`, which designates the first 2000 games for training (no output). Thus, you will only see Pacman play the last 10 of these games. The number of training games is also passed to your agent as the option `numTraining`.
Note: If you want to watch 10 training games to see what's going on, use the command:

```
python pacman.py -p PacmanQAgent -n 10 -l smallGrid -a numTraining=10
```

During training, you will see output every 100 games with statistics about how Pacman is faring. Epsilon is positive during training, so Pacman will play poorly even after having learned a good policy: this is because he occasionally makes a random exploratory move into a ghost. As a benchmark, it should take between 1,000 and 1400 games before Pacman's rewards for a 100-episode segment becomes positive, reflecting that he's started winning more than losing. By the end of the training, it should remain positive and be fairly high (between 100 and 350).

Make sure you understand what is happening here: the MDP state is the exact board configuration facing Pacman, with the now complex transitions describing an entire ply of change to that state. The intermediate game configurations in which Pacman has moved but the ghosts have not replied are not MDP states, but are bundled into the transitions.

Once Pacman is done training, he should win very reliably in test games (at least 90% of the time), since now he is exploiting his learned policy.

However, you will find that training the same agent on the seemingly simple `mediumGrid` does not work well. In our implementation, Pacman's average training rewards remain negative throughout training. At test time, he plays badly, probably losing all of his test games. Training will also take a long time, despite its ineffectiveness.

Pacman fails to win on larger layouts because each board configuration is a separate state with separate Q-values. He has no way to generalize that running into a ghost is bad for all positions. Obviously, this approach will not scale.

5 Question 10 (3 points): Approximate Q-Learning

Implement an approximate Q-learning agent that learns weights for features of states, where many states might share the same features. Write your implementation in `ApproximateQAgent` class in `qlearningAgents.py`, which is a subclass of `PacmanQAgent`.

Note: Approximate Q-learning assumes the existence of a feature function $f(s, a)$ over state and action pairs, which yields a vector $f_1(s, a) \ldots f_i(s, a) \ldots f_n(s, a)$ of feature values. We provide feature functions for you in `featureExtracto`

Feature vectors are `util.Counter` (like a dictionary) objects containing the non-zero pairs of features and values; all omitted features have a value of zero.

The approximate Q-function takes the following form

$$Q(s, a) = \sum_{i=1}^{n} f_i(s, a) w_i$$

where each weight w_i is associated with a particular feature $f_i(s, a)$. In your code, you should implement the weight vector as a dictionary mapping features (which the feature extractors will return) to weight values. You will update your weight vectors similarly to how you updated Q-values:

$$w_i \leftarrow w_i + \alpha \cdot difference \cdot f_i(s, a)$$

$$difference = (r + \gamma \max_{a'} Q(s', a')) - Q(s, a)$$

Note that the $difference$ term is the same as in normal Q-learning, and r is the experience reward.

By default, `ApproximateQAgent` uses the `IdentityExtractor`, which assigns a single feature to every `(state,actio` pair. With this feature extractor, your approximate Q-learning agent should work identically to `PacmanQAgent`. You can test this with the following command:

```
python pacman.py -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid
```

Important: `ApproximateQAgent` is a subclass of `QLearningAgent`, and it therefore shares several methods like `getAction`. Make sure that your methods in `QLearningAgent` call `getQValue` instead of accessing Q-values directly, so that when you override `getQValue` in your approximate agent, the new approximate q-values are used to compute actions.

Once you are confident that your approximate learner works correctly with the identity features, run your approximate Q-learning agent with our custom feature extractor, which can learn to win with ease:

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l
   mediumGrid
```

Even much larger layouts should be no problem for your `ApproximateQAgent`. (*warning:* this may take a few minutes to train)

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l
   mediumClassic
```

If you have no errors, your approximate Q-learning agent should win almost every time with these simple features, even with only 50 training games.
Grading: We will run your approximate Q-learning agent and check that it learns the same Q-values and feature weights as our reference implementation when each is presented with the same set of examples. To grade your implementation, run the autograder:

```
python autograder.py -q q10
```

Congratulations! You have a learning Pacman agent!

6 Evaluation

Once you are done with the tasks, call your course teacher and show them the autograder results and codes.

7 Submission

Submit one file: `StudentID_L6.pdf` (`StudentID` will be replaced by your student ID) under **Assignment 6** on **Google Classroom**. The report can contain (but is not limited to) **introduction** (briefly introduce the problem you tackled, highlight the specific algorithm/technique you implemented), **problem analysis** (discuss the nuances of the problem, explain what modification or unique aspects you were asked to introduce), **solution explanation** (provide an overview of your approach, include portions of the code only where necessary for clarification, highlight key decisions you made during the implementation, analyze the complexity of your code), **findings and insights** (share any interesting observations/insights you gained, discuss how your solution performed in various scenarios), **challenges faced** (describe any challenges you encountered, explain how you overcame these challenges), **hyperparameter exploration** (discuss the impact of different hyperparameter values on your solution's behavior, analyze how changes influenced the performance), **additional information** (feel free to include any extra details, such as extensions, additional features, or creative elements you added to your solution). The report is about insights, analysis, and reflection — no need to duplicate your entire code. Be clear and concise in your explanations. This is your chance to showcase your understanding and creativity, so make the most of it.
You will have **2 weeks** to submit the file. Plagiarism is strictly prohibited and will result in significant penalties.