COURSE: "ARTIFICIAL INTELLIGENT SYSTEMS"
LABORATORY WORK #1
PYTHON FOR ARTIFICIAL INTELLIGENCE. INTRODUCTION.

1.1 Why Python?

We use Python because Python programs can be close to pseudo-code. It is designed for humans to read.

Python is reasonably efficient. Efficiency is usually not a problem for small examples. If your Python code is not efficient enough, a general procedure to improve it is to find out what is taking most the time, and implement just that part more efficiently in some lower-level language. Most of these lower level languages interoperate with Python nicely. This will result in much less programming and more efficient code (because you will have more time to optimize) than writing everything in a low-level language. You will not have to do that for the code here if you are using it for course projects.

1.2 Getting Python

You need **Python 3** (*http://python.org/*) and **matplotlib** (*http://matplotlib.org/*) that runs with **Python 3**. This code is not compatible with **Python 2** (e.g., with **Python 2.7**).

Download and install the latest **Python 3** release from *http://python.org/*.
This should also install **pip3.** You can install **matplotlib** using

pip3 install matplotlib

in a terminal shell (not in Python). That should "just work". If not, try using **pip** instead of **pip3**.

The command python or python3 should then start the interactive python shell. You can quit Python with a control-D or with quit().

To upgrade matplotlib to the latest version (which you should do if you install a new version of Python) do:

pip3 install --upgrade matplotlib

We recommend using the enhanced interactive python **ipython** (http://ipython.org/). To install ipython after you have installed python do:

pip3 install ipython

1.3 Running Python

We assume that everything is done with an interactive Python shell. You can either do this with an IDE, such as IDLE that comes with standard Python distributions, or just running ipython3 (or perhaps just ipython) from a shell.

Here we describe the most simple version that uses no IDE. If you download the zip file, and cd to the "aipython" folder where the .py files are, you should be able to do the following, with user input following : . The first ipython3 command is in the operating system shell (note that the -i is important to enter interactive mode), with user input in bold:

ipython -i searchGeneric.py
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 05:52:31)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.2.1 -- An enhanced Interactive Python. Type '?' for help.
Testing problem 1:
7 paths have been expanded and 4 paths remain in the frontier
Path found: a --> b --> c --> d --> g
Passed unit test

In [1]: **searcher2 = AStarSearcher(searchProblem.acyclic_delivery_problem) #A***

In [2]: **searcher2.search() # find first path**
16 paths have been expanded and 5 paths remain in the frontier
Out[2]: o103 --> o109 --> o119 --> o123 --> r123

In [3]: **searcher2.search() # find next path**
21 paths have been expanded and 6 paths remain in the frontier
Out[3]: o103 --> b3 --> b4 --> o109 --> o119 --> o123 --> r123

In [4]: **searcher2.search() # find next path**
28 paths have been expanded and 5 paths remain in the frontier
Out[4]: o103 --> b3 --> b1 --> b2 --> b4 --> o109 --> o119 --> o123 --> r123

In [5]: **searcher2.search() # find next path**
No (more) solutions. Total of 33 paths expanded.

In [6]:
You can then interact at the last prompt.

There are many textbooks for Python. The best source of information about python is https://www.python.org/. We will be using Python 3; please download the latest release. The documentation is at https://docs.python.org/3/.

The rest of this lab.work is about what is special about the code for AI tools.

We will only use the Standard Python Library and matplotlib. All of the exercises can be done (and should be done) without using other libraries; the aim is for you to spend your time thinking about how to solve the problem rather than searching for pre-existing solutions.

1.4 Pitfalls

It is important to know when side effects occur. Often AI programs consider what would happen or what may have happened. In many such cases, we don't want side effects. When an agent acts in the world, side effects are appropriate.

In Python, you need to be careful to understand side effects. For example, the inexpensive function to add an element to a list, namely append, changes the list. In a functional language like Haskell or Lisp, adding a new element to a list, without changing the original list, is a cheap operation. For example if x is a list containing n elements, adding an extra element to the list in Python (using append) is fast, but it has the side effect of changing the list x. To construct a new list that contains the elements of x plus a new element, without changing the value of x, entails copying the list, or using a different representation for lists.

In the searching code, we will use a different representation for lists for this reason.

1.5 Features of Python

1.5.1 Lists, Tuples, Sets, Dictionaries and Comprehensions

We make extensive uses of lists, tuples, sets and dictionaries (dicts). See https://docs.python.org/3/library/stdtypes.html

One of the nice features of Python is the use of list comprehensions (and also tuple, set and dictionary comprehensions).

```
(fe for e in iter if cond)
```

enumerates the values *fe* for each *e* in *iter* for which *cond* is true. The "if cond" part is optional, but the "for" and "in" are not optional. Here *e* has to be a variable, *iter* is an iterator, which can generate a stream of data, such as a list, a set, a range object (to enumerate integers between ranges) or a file. *cond* is an expression that evaluates to either True or False for each *e*, and *fe* is an expression that will be evaluated for each value of *e* for which *cond* returns *True*.

The result can go in a list or used in another iteration, or can be called directly using next. The procedure next takes an iterator returns the next element (advancing the iterator) and raises a StopIteration exception if there is no next element. The following shows a simple example, where user input is prepended with >>>

```
>>> [e*e for e in range(20) if e%2==0]
[0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
>>> a = (e*e for e in range(20) if e%2==0)
>>> next(a)
0
>>> next(a)
4
>>> next(a)
16
>>> list(a)
[36, 64, 100, 144, 196, 256, 324]
>>> next(a)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration
```

Notice how *list(a)* continued on the enumeration, and got to the end of it.

Comprehensions can also be used for dictionaries. The following code creates an index for list a:

```
>>> a = ["a","f","bar","b","a","aaaaa"]
>>> ind = {a[i]:i for i in range(len(a))}
>>> ind
{'a': 4, 'f': 1, 'bar': 2, 'b': 3, 'aaaaa': 5}
>>> ind['b']
3
```

which means that 'b' is the 3rd element of the list.

The assignment of ind could have also be written as:

```
>>> ind = {val:i for (i,val) in enumerate(a)}
```

where enumerate returns an iterator of (index, value) pairs.

1.5.2 Functions as first-class objects

Python can create lists and other data structures that contain functions. There is an issue that tricks many newcomers to Python. For a local variable in a function, the function uses the last value of the variable when the function is called, not the value of the variable when the function was defined (this is called “late binding”). This means if you want to use the value a variable has when the function is created, you need to save the current value of that variable.

Whereas Python uses “late binding” by default, the alternative that newcomers often expect is “early binding”, where a function uses the value a variable had when the function was defined, can be easily implemented.

Consider the following programs designed to create a list of 5 functions, where the ith function in the list is meant to add i to its argument:

pythonDemo.py | Some tricky examples

```
11 fun_list1 = []
12 for i in range(5):
13 def fun1(e):
14 return e+i
15 fun_list1.append(fun1)
16
17 fun_list2 = []
18 for i in range(5):
19 def fun2(e,iv=i):
20 return e+iv
21 fun_list2.append(fun2)
22
23 fun_list3 = [lambda e: e+i for i in range(5)]
24
25 fun_list4 = [lambda e,iv=i: e+iv for i in range(5)]
26
27 i=56
```

Try to predict, and then test to see the output, of the output of the following calls, remembering that the function uses the latest value of any variable that is not bound in the function call:

pythonDemo.py | (continued)

```
29 # in Shell do
30 ## ipython -i pythonDemo.py
31 # Try these (copy text after the comment symbol and paste in the Python prompt):
32 # print([f(10) for f in fun_list1])
33 # print([f(10) for f in fun_list2])
34 # print([f(10) for f in fun_list3])
35 # print([f(10) for f in fun_list4])
```

In the first for-loop, the function fun uses i, whose value is the last value it was assigned. In the second loop, the function fun2 uses iv. There is a separate *iv* variable for each function, and its value is the value of i when the function was defined. Thus fun1 uses late binding, and fun2 uses early binding. *fun list3* and *fun list4* are equivalent to the first two (except fun list4 uses a different I variable).

One of the advantages of using the embedded definitions (as in fun1 and fun2 above) over the lambda is that is it possible to add a __doc__ string, which is the standard for documenting functions in Python, to the embedded definitions.

1.5.3 Generators and Coroutines

Python has generators which can be used for a form of coroutines.

The ***yield*** command returns a value that is obtained with next. It is typically used to enumerate the values for a for loop or in generators.

A version of the built-in range, with 2 or 3 arguments (and positive steps) can be implemented as:

pythonDemo.py | (continued)

```
37 def myrange(start, stop, step=1):
38 """enumerates the values from start in steps of size step that
39 are less than stop.
40 """
41 assert step>0, "only positive steps implemented in myrange"
42 i = start
43 while i<stop:
44 yield i
45 i += step
46
47 print("myrange(2,30,3):",list(myrange(2,30,3)))
```

Note that the built-in range is unconventional in how it handles a single argument, as the single argument acts as the second argument of the function. Note also that the built-in range also allows for indexing (e.g., range(2, 30, 3)[2] returns 8), which the above implementation does not. However myrange also works for floats, which the built-in range does not.

Exercise 1.1 Implement a version of myrange that acts like the built-in version when there is a single argument. (Hint: make the second argument have a default value that can be recognized in the function.)

Yield can be used to generate the same sequence of values as in the example of Section 1.5.1:

pythonDemo.py | (continued)

```
49 def ga(n):
50  """generates square of even nonnegative integers less than
n"""
51 for e in range(n):
52 if e%2==0:
53 yield e*e
54 a = ga(20)
```

The sequence of next(a), and list(a) gives exactly the same results as the comprehension in Section 1.5.1.

It is straightforward to write a version of the built-in enumerate. Let's call it myenumerate:

pythonDemo.py | (continued)

```
56 def myenumerate(enum):
57 for i in range(len(enum)):
58 yield i,enum[i]
```

Exercise 1.2 Write a version of enumerate where the only iteration is "for val in enum". Hint: keep track of the index.

1.6 Useful Libraries

1.6.1 Timing Code

In order to compare algorithms, we often want to compute how long a program takes; this is called the **runtime** of the program. The most straightforward way to compute runtime is to use time.perf counter(), as in:

```
import time
start_time = time.perf_counter()
compute_for_a_while()
end_time = time.perf_counter()
print("Time:", end_time - start_time, "seconds")
```

Note that time.perf_counter() measures clock time; so this should be done without user interaction between the calls. On the console, you should do:

```
start_time = time.perf_counter(); compute_for_a_while(); end_time = time.perf_counter()
```

If this time is very small (say less than 0.2 second), it is probably very inaccurate, and it may be better to run your code many times to get a more accurate count. For this you can use timeit (https://docs.python.org/3/library/timeit.html). To use timeit to time the call to foo.bar(aaa) use:

```
import timeit
time = timeit.timeit("foo.bar(aaa)",
setup="from __main__ import foo,aaa", number=100)
```

The setup is needed so that Python can find the meaning of the names in the string that is called. This returns the number of seconds to execute foo.bar(aaa) 100 times. The variable number should be set so that the runtime is at least 0.2 seconds.

You should not trust a single measurement as that can be confounded by interference from other processes. timeit.repeat can be used for running *timit* a few (say 3) times. Usually the minimum time is the one to report, but you should be explicit and explain what you are reporting.

1.6.2 Plotting: Matplotlib

The standard plotting for Python is matplotlib (http://matplotlib.org/). We will use the most basic plotting using the pyplot interface.

Here is a simple example that uses everything we will use.

pythonDemo.py | (continued)

```
60 import matplotlib.pyplot as plt
61
62 def myplot(min,max,step,fun1,fun2):
63 plt.ion() # make it interactive
64 plt.xlabel("The x axis")
65 plt.ylabel("The y axis")
66 plt.xscale('linear') # Makes a 'log' or 'linear' scale
67 xvalues = range(min,max,step)
68 plt.plot(xvalues,[fun1(x) for x in xvalues],
69 label="The first fun")
70  plt.plot(xvalues,[fun2(x)  for  x  in  xvalues],  linestyle='--
',color='k',
71 label=fun2.__doc__) # use the doc string of the function
72 plt.legend(loc="upper right") # display the legend
73
74 def slin(x):
75 """y=2x+7"""
76 return 2*x+7
77 def sqfun(x):
78 """y=(x-40)^2/10-20"""
79 return (x-40)**2/10-20
80
81 # Try the following:
82 # from pythonDemo import myplot, slin, sqfun
83 # import matplotlib.pyplot as plt
84 # myplot(0,100,1,slin,sqfun)
85 # plt.legend(loc="best")
86 # import math
87 # plt.plot([41+40*math.cos(th/10) for th in range(50)],
88 # [100+100*math.sin(th/10) for th in range(50)])
89 # plt.text(40,100,"ellipse?")
90 # plt.xscale('log')
```

At the end of the code are some commented-out commands you should try in interactive mode. Cut from the file and paste into Python (and remember to remove the comments symbol and leading space).

1.7 Utilities

1.7.1 Display

In this distribution, to keep things simple and to only use standard Python, we use a text-oriented tracing of the code. A graphical depiction of the code could override the definition of display (but we leave it as a project).

The method self .display is used to trace the program. Any call

```
self .display(level, to print . . . )
```

where the level is less than or equal to the value for max display level will be printed. The to print . . . can be anything that is accepted by the built-in print (including any keyword arguments).

The definition of display is:

display.py | A simple way to trace the intermediate steps of algorithms.

```
11 class Displayable(object):
12 """Class that uses 'display'.
13 The amount of detail is controlled by max_display_level
14 """
15 max_display_level = 1 # can be overridden in subclasses
16
17 def display(self,level,*args,**nargs):
18 """print the arguments if level is less than or equal to the
19 current max_display_level.
20 level is an integer.
21 the other arguments are whatever arguments print can take.
22 """
23 if level <= self.max_display_level:
24  print(*args, **nargs) ##if error you are using Python2 not
Python3
```

Note that args gets a tuple of the positional arguments, and nargs gets a dictionary of the keyword arguments). This will not work in Python 2, and will give an error.

Any class that wants to use display can be made a subclass of Displayable.

To change the maximum display level to say 3, for a class do:

```
Classname.max display level = 3
```

which will make calls to display in that class print when the value of level is less than-or-equal to 3. The default display level is 1. It can also be changed for individual objects (the object value overrides the class value).

The value of max display level by convention is:

0 display nothing
1 display solutions (nothing that happens repeatedly)
2 also display the values as they change (little detail through a loop)
3 also display more details
4 and above even more detail

In order to implement more sophisticated visualizations of the algorithm, we add a **visualize** "decorator" to the methods to be visualized. The following code ignores the decorator:

display.py | (continued)

```
26 def visualize(func):
27 """A decorator for algorithms that do interactive
28 visualization. Ignored here.
29 """
30 return func
```

1.7.2 Argmax

Python has a built-in max function that takes a generator (or a list or set) and returns the maximum value. The argmax method returns the index of an element that has the maximum value. If there are multiple elements with the maximum value, one if the indexes to that value is returned at random. argmaxe assumes an enumeration; a generator of (element, value) pairs, as for example is generated by the built-in enumerate(list) for lists or dict.items() for dicts.

utilities.py | AIPython useful utilities

```
11 import random
12 import math
13
14 def argmaxall(gen):
15 """gen is a generator of (element,value) pairs, where value is
a real.
16 argmaxall returns a list of all of the elements with maximal
value.
17 """
18 maxv = -math.inf # negative infinity
19 maxvals = [] # list of maximal elements
20 for (e,v) in gen:
21 if v>maxv:
22 maxvals,maxv = [e], v
23 elif v==maxv:
24 maxvals.append(e)
25 return maxvals
26
```

```
27 def argmaxe(gen):
28 """gen is a generator of (element,value) pairs, where value is a real.
29 argmaxe returns an element with maximal value.
30 If there are multiple elements with the max value, one is returned at random.
31 """
32 return random.choice(argmaxall(gen))
33
34 def argmax(lst):
35 """returns maximum index in a list"""
36 return argmaxe(enumerate(lst))
37 # Try:
38 # argmax([1,6,3,77,3,55,23])
39
40 def argmaxd(dct):
41 """returns the arx max of a dictionary dct"""
42 return argmaxe(dct.items())
43 # Try:
44 # arxmaxd({2:5,5:9,7:7})
```

Exercise 1.3 Change argmax to have an optional argument that specifies whether you want the "first", "last" or a "random" index of the maximum value returned. If you want the first or the last, you don't need to keep a list of the maximum elements.

1.7.3 Probability

For many of the simulations, we want to make a variable True with some probability. flip(p) returns True with probability p, and otherwise returns False.

utilities.py | (continued)

```
45 def flip(prob):
46 """return true with probability prob"""
47 return random.random() < prob
```

1.7.4 Dictionary Union

This is now | in Python 3.9, so will be replaced.

The function dict union(d1, d2) returns the union of dictionaries d1 and d2. If the values for the keys conflict, the values in d2 are used. This is similar to dict(d1, _ _ d2), but that only works when the keys of d2 are strings.

utilities.py | (continued)

```
49 def dict_union(d1,d2):
50 """returns a dictionary that contains the keys of d1 and d2.
51 The value for each key that is in d2 is the value from d2,
```

```
52 otherwise it is the value from d1.
53 This does not have side effects.
54 """
55 d = dict(d1) # copy d1
56 d.update(d2)
57 return d
```

1.8 Testing Code

It is important to test code early and test it often. We include a simple form of **unit test**. The value of the current module is in __name__ and if the module is run at the top-level, it's value is "__main__". See https://docs.python.org/3/library/ main .html.

The following code tests argmax and dict_union, but only when if utilities is loaded in the top-level. If it is loaded in a module the test code is not run.

In your code you should do more substantial testing than we do here, in particular testing the boundary cases.

utilities.py | *(continued)*

```
59 def test():
60 """Test part of utilities"""
61 assert argmax(enumerate([1,6,55,3,55,23])) in [2,4]
62  assert dict_union({1:4, 2:5, 3:4},{5:7, 2:9}) == {1:4, 2:9,
3:4, 5:7}
63 print("Passed unit test in utilities")
64
65 if __name__ == "__main__":
66 test()
```

QUIZ:

1. Find the index of the 1st matching element.
2. What is the difference between a list and a tuple? Create both of them.
3. What is the difference between a list and a set? Create both of them.
4. Insert a value at a specific index in an existing list
5. Convert a list into a dictionary where list elements are keys.
   For this we can use a dictionary comprehension.

```
li = ['The', 'quick', 'brown', 'fox', 'was', 'quick']
d = {k:1 for k in li}
d #=> {'The': 1, 'quick': 1, 'brown': 1, 'fox': 1, 'was': 1}
```

6. Remove elements in a list after a specific index.

7. Using the slice syntax, we can return a new list with only the elements up to a specific index.

```
li = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,10]
li[:10]
#=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

8. Return every 2nd element in a list between 2 indices.
Or before/after/between indices at a specific interval.
Here we return every 2nd value between the indices 10 and 16 using the slice syntax.

```
li = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,10]
li[10:16:2]
#=> [11, 13, 15]
```

9. Find the intersection of 2 lists.
We can do this by utilizing `set()` with an ampersand.

```
li1 = [1,2,3]
li2 = [2,3,4]set(li1) & set(li2)
#=> {2, 3}
```

10. Combine 2 lists into a dictionary
Using `zip()` and the `list()` constructor we can combine 2 lists into a dictionary where one list becomes the keys and the other list becomes the values.

```
name = ['Snowball', 'Chewy', 'Bubbles', 'Gruff']
animal = ['Cat', 'Dog', 'Fish', 'Goat']dict(zip(name, animal))
#=> {'Snowball': 'Cat', 'Chewy': 'Dog', 'Bubbles': 'Fish',
'Gruff': 'Goat'}
```