**Artificial Intelligence**

CSL-411

Lab Manual



**Fall 2021**

**Department of Computer Science, Bahria University, Islamabad, Campus**

**List of Experiments:**

|  |  |
| --- | --- |
| **Lab. No** | **Experiment Names** |
| 1 | Introduction to Python (& Installation), Syntax, Basic Functions, Control Structures, Loops & Functions |
| 2 | Lists, Tuples, Sets & Dictionary, Classes & Inheritance, Modules in Python & Examples |
| 3 | Simple Reflex Agents, Model based Reflex Agents |
| 4 | Graph in Python |
| 5 | Uninformed Search |
| 6 | Searching Algorithms: Informed/Heuristic Search |
| 7 | Game Search |
| 8 | Constraint Satisfaction Problems |
| 9 | Tkinter (GUI) |
| 10 | Machine Learning |
| 11 | Introduction to Prolog, Installation of SWIProlog/Prolog software First Order Logic |
| 12 | Forward Chaining, Backward Chaining, Expert Systems |
| 13 | File Handling & Regular Expression |
| 14 | Revision of Concepts/Project Work |

Lab 1-Part 1

**Installing Python:**

[www.python.org](http://www.python.org/)

For Windows (32 bit): <https://www.python.org/ftp/python/3.4.3/python-3.4.3.msi>

For Windows (64 bit): <https://www.python.org/ftp/python/3.4.3/python-3.4.3.amd64.msi>

**Opening IDLE**

Go to the start menu, find Python, and run the program labeled 'IDLE'

(Stands for Integrated DeveLopment Environment)

**Code Example 1 - Hello, World!**

>>> print ("Hello, World!" )

**Learning python for a C++/C# programmer**

Let us try to quickly compare the syntax of python with that of C++/C#:

|  |  |  |
| --- | --- | --- |
|  | **C++/C#** | **Python** |
| Comment begins with | // | # |
| Statement ends with | ; | No semi-colon needed |
| Blocks of code | Defined by {} | Defined by indentation (usually four spaces) |
| Indentation of code and use of white space | Is irrelevant | Must be same for same block of code (for example for a set of statements to be executed after a particular if statement) |
| Conditional statement | if-else if- else | if – elif – else: |
| Parentheses for loop execution condition | Required | Not required but loop condition followed by a colon :  while a < n:  print(a) |

Lab 1-Part 2

**Math in Python**

Calculations are simple with Python, and expression syntax is straightforward: the operators +, -, \* and / work as expected; parentheses () can be used for grouping.

# Python 3: Simple arithmetic

>>> 1 / 2

0.5

>>> 2 \*\* 3 #Exponent operator

8

>>> 17 / 3 # classic division returns a float

5.666666666666667

>>> 17 // 3 # floor division

5

>>> 23%3 #Modulus operator

2

**Python Operators**

|  |  |  |  |
| --- | --- | --- | --- |
| **Command** | **Name** | **Example** | **Output** |
| + | Addition | 4+5 | 9 |
| - | Subtraction | 8-5 | 3 |
| \* | Multiplication | 4\*5 | 20 |
| / | Classic Division | 19/3 | 6.3333 |
| % | Modulus | 19%3 | 5 |
| \*\* | Exponent | 2\*\*4 | 16 |
| // | Floor Division | 19/3 | 6 |

**Comments in Python:**

#I am a comment. I can say whatever I want!

**Variables:**

print ("This program is a demo of variables")

v = 1

print ("The value of v is now", v)

v = v + 1

print ("v now equals itself plus one, making it worth", v)

print ("To make v five times bigger, you would have to type v = v \* 5")

v = v \* 5

print ("There you go, now v equals", v, "and not", v / 5 )

**Strings:**

word1 = "Good"

word2 = "Morning"

word3 = "to you too!"

print (word1, word2)

sentence = word1 + " " + word2 + " " +word3

print (sentence)

**Relational operators:**

|  |  |
| --- | --- |
| Expression | Function |
| < | less than |
| <= | less than or equal to |
| > | greater than |
| >= | greater than or equal to |
| != | not equal to |
| == | is equal to |

**Boolean Logic:**

Boolean logic is used to make more complicated conditions for **if** statements that rely on more than one condition. Python’s Boolean operators are **and**, **or**, and **not**. The **and** operator takes two arguments, and evaluates as **True** if, and only if, both of its arguments are True. Otherwise it evaluates to **False**.

The **or** operator also takes two arguments. It evaluates if either (or both) of its arguments are **False**.

Unlike the other operators we’ve seen so far, **not** only takes one argument and inverts it. The result of **not True** is **False**, and **not False** is **True**.

**Operator Precedence:**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| () | Parentheses |
| \*\* | Exponentiation (raise to the power) |
| ~ + - | Complement, unary plus and minus |
| \* / % // | Multiply, divide, modulo, and floor division |
| + - | Addition and subtraction |
| >> << | Right and left bitwise shift |
| & | Bitwise ‘AND’ |
| ^ | | Bitwise exclusive ‘OR’ and regular ‘OR’ |
| <= < > >= | Comparison Operators |
| == != | Equality Operators |
| = %= /= //= -= += \*= \*\*= | Assignment operators |
| is is not | Identity operators |
| in not in | Membership operators |
| not or and | Logical operators |

**Conditional Statements:**

**‘if' - Statement**

y = 1

if y == 1:

print ("y still equals 1, I was just checking")

**‘if - else' - Statement**

a = 1

if a > 5:

print ("This shouldn't happen.")

else:

print ("This should happen.")

**‘elif' - Statement**

z = 4

if z > 70:

print ("Something is very wrong")

elif z < 7:

print ("This is normal")

**LAB TASK:**

1. Open IDLE and run the following program. Try different integer values for separate runs of the program. Play around with the indentation of the program lines of code and run it again. See what happens. Make a note of what changes you made and how it made the program behave. Also note any errors, as well as the changes you need to make to remove the errors.

x = input("Please enter an integer: ")

if x < 0:

x = 0

print('Negative changed to zero')

elif x == 0:

print('Zero')

elif x == 1:

print('Single')

else:

print('More')

Lab 1- Part 3

**Input from user:**

The **input()** function prompts for input and returns a string.

a = input (“Enter Value for variable a: ”)

print (a)

**Indexes of String:**

Characters in a string are numbered with *indexes* starting at 0:

Example:

name = "J. Smith“

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Index | 00 | 11 | 22 | 33 | 44 | 55 | 66 | 77 |
| Character | pJ | .. |  | AS | im | Si | Ht | ah |

Accessing an individual character of a string:

**variableName** [ **index** ]

Example:

print (name, " starts with", name[0])

Output:

J. Smith starts with J

**input:**

input: Reads a string of text from user input.

Example:

name = input("What's your name? ")

print (name, "... what a nice name!")

Output:

What's your name? Ali

Ali... what a nice name!

**String Properties:**

len(*string*) - number of characters in a string (including spaces)

str.lower(*string*) - lowercase version of a string

str.upper(*string*) - uppercase version of a string

Example:

name = "Linkin Park"

length = len(name)

big\_name = str.upper(name)

print (big\_name, "has", length, "characters")

Output:

LINKIN PARK has 11 characters

**Strings and numbers:**

ord(*text*) - converts a string into a number.

Example: ord(‘a’) is 97, ord("b") is 98, ...

Characters map to numbers using standardized mappings such as *ASCII* and *Unicode*.

chr (*number*) - converts a number into a string.

Example: chr(99) is "c"

**Loops in Python:**

**The 'while' loop**

a = 0

while a < 10:

a = a + 1

print (a )

**The 'for' loop**

for i in range(1, 5):

print (i )

for i in range(1, 5):

print (i)

else:

print ('The for loop is over')

**Functions:**

**How to call a function?**

function\_name(parameters)

Code Example - Using a function

def multiplybytwo(x):

return x\*2

a = multiplybytwo(70)

The computer would actually see this:

a=140

**Define a Function?**

def function\_name(parameter\_1,parameter\_2):

{this is the code in the function}

return {value (e.g. text or number) to return to the n program}

range() **Function:**

If you need to iterate over a sequence of numbers, the built-in function range() comes in handy. It generates iterator containing arithmetic progressions:

>>> range(10) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

>>> list(range(5, 10) )

[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3) )

[0, 3, 6, 9]

>>> list(range(-10, -100, -30) )

[-10, -40, -70]

The range() function is especially useful in loops.

**LAB TASK:**

1. Write a simple calculator program. Follow the steps below:
   1. Declare and define a function named Menu which displays a list of choices for user such as addition, subtraction, multiplication, & classic division. It takes the choice from user as an input and return.
   2. Define and declare a separate function for each choice.
   3. In the main body of the program call respective function depending on user’s choice.
   4. Program should not terminate till user chooses option to “Quit”.
2. Implement the following functions for the calculator you created in the above task. (Task 2)
   1. Factorial
   2. x\_power\_y (x raised to the power y)
   3. log
   4. ln (Natural log)

Lab 2- Part 1

**Classes & Inheritance:**

The word 'class' can be used when describing the code where the class is defined.

A variable inside a class is known as an *Attribute*

A function inside a class is known as a *method*

* A class is like a
  + Prototype
  + Blue-print
  + An object creator
* A class defines potential objects
  + What their structure will be
  + What they will be able to do
* Objects are instances of a class
  + An object is a container of data: attributes
  + An object has associated functions: methods

**Syntax:**

# Defining a class

class class\_name:

[statement 1]

[statement 2]

[statement 3] [etc]

**Inheritance Syntax:**

class child\_class(parent\_class):

def \_\_init\_\_(self,x):

# it will modify the \_init\_ function from parent class

# additional methods can be defined here

**‘self’ keyword:**

The first argument of every class method, including \_\_init\_\_, is always a reference to the current instance of the class. By convention, this argument is always named self. In the \_\_init\_\_ method, self refers to the newly created object; in other class methods, it refers to the instance whose method was called.

**Example1:**

class MyClass:

i = 12345

def f(self):

return 'hello world'

x = MyClass()

print (x.i)

print (x.f() )

**Example2:**

class Complex:

def \_\_init\_\_(self, realpart, imagpart):

self.r = realpart

self.i = imagpart

x = Complex(3.0, -4.5)

print (x.r," ",x.i )

**Example3:**

class Shape:

def \_\_init\_\_(self,x,y): #The \_\_init\_\_ function always runs first

self.x = x

self.y = y

description = "This shape has not been described yet"

author = "Nobody has claimed to make this shape yet"

def area(self):

return self.x \* self.y

def perimeter(self):

return 2 \* self.x + 2 \* self.y

def describe(self,text):

self.description = text

def authorName(self,text):

self.author = text

def scaleSize(self,scale):

self.x = self.x \* scale

self.y = self.y \* scale

a=Shape(3,4)

print (a.area())

**Inheritance Example:**

class Square(Shape):

def \_\_init\_\_(self,x):

self.x = x

self.y = x

class DoubleSquare(Square):

def \_\_init\_\_(self,y):

self.x = 2 \* y

self.y = y

def perimeter(self):

return 2 \* self.x + 2 \* self.y

**Module:**

A module is a python file that (generally) has only definitions of variables, functions, and classes.

**Example:** Module name mymodule.py

# Define some variables:

ageofqueen = 78

# define some functions

def printhello():

print ("hello")

# define a class

class Piano:

def \_\_init\_\_(self):

self.type = input("What type of piano?: ")

self.height = input("What height (in feet)?: ")

self.price = input("How much did it cost?: ")

self.age = input("How old is it (in years)?: ")

def printdetails(self):

print ("This piano is a/an " + self.height + " foot")

print (self.type, "piano, " + self.age, "years old and costing " + self.price + " dollars.")

**Importing module in main program:**

### mainprogam.py ##

# IMPORTS ANOTHER MODULE

import mymodule

print (mymodule.ageofqueen )

cfcpiano = mymodule.Piano()

cfcpiano.printdetails()

Another way of importing the module is:

from mymodule import Piano, ageofqueen

print (ageofqueen)

cfcpiano = Piano()

cfcpiano.printdetails()

**Lab Task 2A:**

1. Create a class name basic\_calc with following attributes and methods;

Two integers (values are passed with instance creation)

Different methods such as addition, subtraction, division, multiplication

Create another class inherited from basic\_calc named s\_calc which should have the following additional methods;

Factorial, x\_power\_y,log, ln, sin, cos, tantc

1. Modify the classes created in the above task under as follows:

Create a module name basic.py having the class name basic\_calc with all the attributes and methods defined before.

Now import the basic.py module in your program and do the inheritance step defined before i.e.

Create another class inherited from basic\_calc named s\_calc which should have the following additional methods;

Factorial, x\_power\_y, log, ln etc

Lab 2-Part 2

**Lists:**

Lists are what they seem - a list of values. Each one of them is numbered, starting from zero. You can remove values from the list, and add new values to the end. Example: Your many cats' names. *Compound* data types, used to group together other values. The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets. List items need not all have the same type.

cats = ['Tom', 'Snappy', 'Kitty', 'Jessie', 'Chester']

print (cats[2])

cats.append(‘Oscar’)

#Remove 2nd cat, Snappy.

del cats[1]

**Compound datatype:**

>>> a = ['spam', 'eggs', 100, 1234]

>>> a[1:-1] #start at element at index 1, end before last element

['eggs', 100]

>>> a[:2] + ['bacon', 2\*2]

['spam', 'eggs', 'bacon', 4]

>>> 3\*a[:3] + ['Boo!']

['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']

>>> a= ['spam', 'eggs', 100, 1234]

>>> a[2] = a[2] + 23

>>> a

['spam', 'eggs', 123, 1234]

**Replace some items:**

>>> a[0:2] = [1, 12]

>>> a

[1, 12, 123, 1234]

**Remove some:**

>>> a[0:2] = []

>>> a

[123, 1234]

**Clear the list: replace all items with an empty list:**

>>> a[:] = []

>>> a

[]

**Length of list:**

>>> a = ['a', 'b', 'c', 'd']

>>> len(a)

4

**Nested lists:**

>>> q = [2, 3]

>>> p = [1, q, 4]

>>> len(p)

3

>>> p[1]

[2, 3]

**Functions of lists:**

**list.append(x):** Add an item to the end of the list; equivalent to a[len(a):] = [x].

**list.extend(L):** Extend the list by appending all the items in the given list; equivalent to a[len(a):] = L.

**list.insert(i, x):** Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x).

**list.remove(x):** Remove the first item from the list whose value is x. It is an error if there is no such item.

**list.pop(i):** Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list.

**list.count(x):** Return the number of times x appears in the list.

**list.sort():** Sort the items of the list, in place.

**list.reverse():** Reverse the elements of the list, in place.

**Tuples:**

Tuples are just like lists, but you can't change their values. Again, each value is numbered starting from zero, for easy reference. Example: the names of the months of the year.

months = ('January' , 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November', 'December')

|  |  |
| --- | --- |
| **Index** | **Value** |
| 0 | January |
| 1 | February |
| 2 | March |
| 3 | April |
| 4 | May |
| 5 | June |
| 6 | July |
| 7 | August |
| 8 | September |
| 9 | October |
| 10 | November |
| 11 | December |

We can have easy membership tests in Tuples using the keyword in.

>>> 'December' in months # fast membership testing

True

**Sets:**

A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the set() function can be used to create sets. Note: to create an empty set you have to use set(), not {}; the latter creates an empty dictionary.

Example 1:

>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']

>>> fruit = set(basket) # create a set without duplicates

>>> fruit

{'banana', 'orange', 'pear', 'apple' }

>>> 'orange' in fruit # fast membership testing

True

>>> 'crabgrass' in fruit

False

Example 2:

>>> # Demonstrate set operations on unique letters from two words

>>> a = set('abracadabra')

>>> b = set('alacazam')

>>> a # unique letters in a

{'a', 'r', 'b', 'c', 'd'}

>>> a - b # letters in a but not in b

{'r', 'd', 'b'}

>>> a | b # letters in either a or b

{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}

>>> a & b # letters in both a and b

{'a', 'c'}

>>> a ^ b # letters in a or b but not both

{'r', 'd', 'b', 'm', 'z', 'l'}

Set comprehensions are also supported:

>>> a = {x for x in 'abracadabra' if x not in 'abc'}

>>> a

{'r', 'd'}

**Dictionaries:**

Dictionaries are similar to what their name suggests - a dictionary. In a dictionary, you have an 'index' of words, and for each of them a definition.

In python, the word is called a 'key', and the definition a 'value'. The values in a dictionary aren't numbered - they aren't in any specific order, either - the key does the same thing.

You can add, remove, and modify the values in dictionaries. Example: telephone book.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with del. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Performing list(d.keys()) on a dictionary returns a list of all the keys used in the dictionary, in arbitrary order (if you want it sorted, just use sorted(d.keys()) instead). To check whether a single key is in the dictionary, use the **in** keyword.

At one time, only one value may be stored against a particular key. Storing a new value for an existing key overwrites its old value. If you need to store more than one value for a particular key, it can be done by storing a list as the value for a key.

phonebook = {'ali':8806336, 'omer':6784346,'shoaib':7658344, 'saad':1122345}

#Add the person '' to the phonebook:

phonebook['waqas'] = 1234567

print("Original Phonebook")

print(phonebook)

# Remove the person 'shoaib' from the phonebook:

del phonebook['shoaib']

print("'shoaib' deleted from phonebook")

print(phonebook)

phonebook = {'Andrew Parson':8806336, \

'Emily Everett':6784346, 'Peter Power':7658344, \

'Louis Lane':1122345}

print("New phonebook")

print(phonebook)

#Add the person 'Gingerbread Man' to the phonebook:

phonebook['Gingerbread Man'] = 1234567

list(phonebook.keys())

sorted(phonebook.keys())

print( 'waqas' in phonebook)

print( 'Emily Everett' in phonebook)

#Delete the person 'Gingerbread Man' from the phonebook:

del phonebook['Gingerbread Man']

**Generators:**

Generators are very easy to implement, but a bit difficult to understand. Generators are used to create iterators, but with a different approach. Generators are simple functions which return an iterable set of items, one at a time, in a special way.

When an iteration over a set of item starts using the for statement, the generator is run. Once the generator's function code reaches a "**yield**" statement, the generator yields its execution back to the for loop, returning a new value from the set. The generator function can generate as many values (possibly infinite) as it wants, yielding each one in its turn.

Here is a simple example of a generator function which returns 7 random integers:

import random

def lottery():

# returns 6 numbers between 1 and 40

for i in range(6):

yield random.randint(1, 40)

# returns a 7th number between 1 and 15

yield random.randint(1,15)

for random\_number in lottery():

print ("And the next number is... %d!" % random\_number)

**Lab Task 2B:**

1. Write a program that takes two integers as input (lower limit and upper limit) and displays all the prime numbers including and between these two numbers
2. Create list of Fibonacci numbers after calculating Fibonacci series up to the number n which you will pass to a function as an argument. The number n must be input by the user.

They are calculated using the following formula: The first two numbers of the series is always equal to 1, and each consecutive number returned is the sum of the last two numbers. Hint: Can you use only two variables in the generator function?

a = 1

b = 2

a, b = b, a

will simultaneously switch the values of a and b.

The first number in the series should be 1. (The output will start like 1,1,2,3,5,8, …)

1. Write a program that lets the user enter some English text, then converts the text to Pig-Latin. To review, Pig-Latin takes the first letter of a word, puts it at the end, and appends “ay”. The only exception is if the first letter is a vowel, in which case we keep it as it is and append “hay” to the end. For example: “hello” -> “ellohay”, and “image” -> “imagehay”

It will be useful to define a list or tuple at the top called VOWELS. This way, you can check if a letter *x* is a vowel with the expression x in VOWELS.

It’s tricky for us to deal with punctuation and numbers with what we know so far, so instead, ask the user to enter only words and spaces. You can convert their input from a string to a list of strings by calling split on the string:

“My name is John Smith”.split(“ ”) -> [“My”, “name”, “is”, “John”, “Smith”]

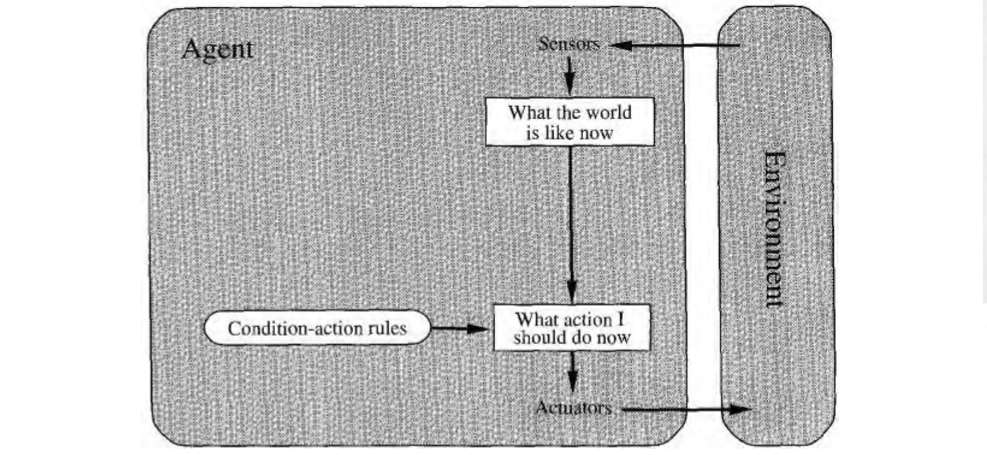
1. Add the following functions to the s\_calc you created in Task 2A-1 and Task2A-2.
   1. sin(x)
   2. cos(x)
   3. tan(x)
   4. sqrt

Lab 3

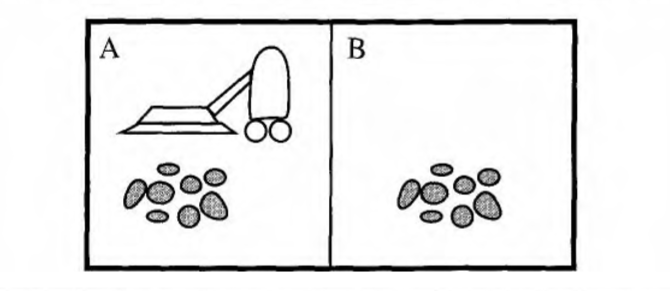
**Agents:**

**Simple Reflex Agents**

Simple reflex agents act only on the basis of the current percept, ignoring the rest of the percept history. The agent function is based on the condition-action rule: if condition then action. This agent function only succeeds when the environment is fully observable. Some reflex agents can also contain information on their current state which allows them to disregard conditions whose actuators are already triggered. A schematic diagram of a simple reflex agent is shown below:



Consider the vacuum world shown in the figure below:



This particular world has just two locations: squares A and B. The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right, suck up the dirt, or do nothing. One very simple **agent function** is the following: if the current square is dirty, then suck, otherwise move to the other square.

Suppose that a simple reflex vacuum agent is deprived of its location sensor, and has only a dirt sensor. Such an agent has just two possible percepts: [Dirty] and [Clean]. It can Suck in response to [Dirty]; what should it do in response to [Clean]? Moving Lefl fails (for ever) if it happens to start in square A, and moving Right fails (for ever) if it happens to start in square B. Infinite loops are often unavoidable for simple reflex agents operating in partially observable environments.

Escape from infinite loops is possible if the agent can **randomize** its actions. For example, if the vacuum agent perceives [Clean], it might flip a coin to choose between Left and Right. It is easy to show that the agent will reach the other square in an average of two steps. Then, if that square is dirty, it will clean it and the cleaning task will be complete. Hence, a randomized simple reflex agent might outperform a deterministic simple reflex agent.

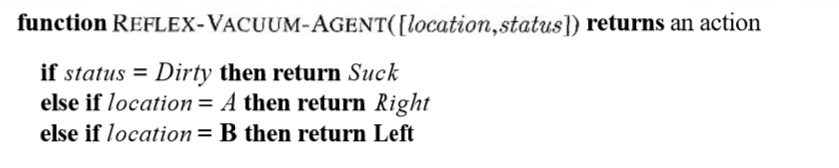
**Performance Measure:**

The performance measure evaluates the behaviour of the agent in an environment. A rational agent acts to maximize the expected value of the performance measure, given the percept sequence it has seen so far.

**Agent program:**

The job of A1 is to design the agent program that implements the agent function mapping percepts to actions. The agent program implements the agent function. There exists a variety of basic agent-program designs, reflecting the kind of information made explicit and used in the decision process. The designs vary in efficiency, compactness, and flexibility. The appropriate design of the agent program depends on the nature of the environment.

A simple program for the agent function of vacuum-world is shown below:



**Example 1**

percept= ["Anum", "Taimur", "Ali", "Saad"]

state = ["happy", "sad", "angry", "normal"]

rule= ["smile", "cry", "frown", "watch football"]

def GetState(cPercept):

index=-1

for p in percept:

index=index+1

if p==cPercept:

return state[index]

def GetRule(cState):

index=-1

for s in state:

index=index+1

if s==cState:

return rule[index]

def SimpleReflexAgent(cPercept):

return GetRule(GetState(cPercept))

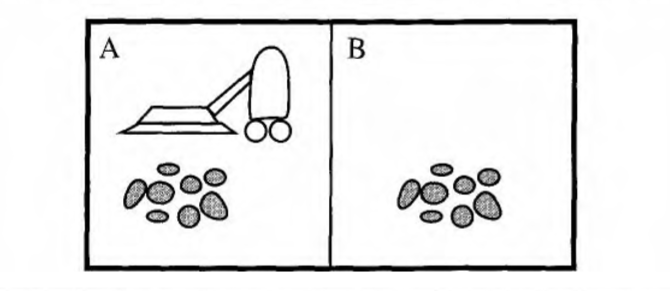
print ("MENU: ")

print (" 0:Anum 1:Taimur 2:Ali 3:Saad")

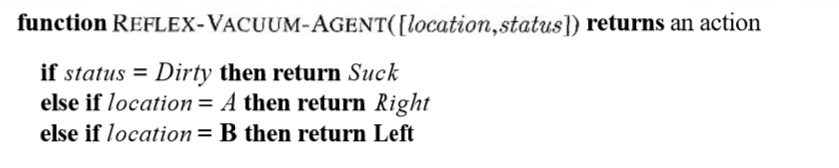
print (SimpleReflexAgent(percept[int(input("Input Number :"))]))

**Lab Tasks:**

* + - 1. Consider the vacuum world shown in the figure below:



This particular world has just two locations: squares A and B. The vacuum agent perceives which square it is in and1 whether there is dirt in the square. It can choose to move left, move right, suck up the dirt, or do nothing. One very simple agent function is the following: if the current square is dirty, then suck, otherwise move to the other square. A simple program for the agent function of vacuum-world is shown below:



Your task is to implement the above vacuum world and its agent program. Also, suggest a performance measure and evaluate your program based on that performance measure.

* + - 1. Model Based Agent: Pacman

Lab 4

**Graphs:**

Consider a simple (directed) graph (digraph) having six nodes (A-F) and the following arcs (directed edges):

A -> B

A -> C

B -> C

B -> D

C -> D

C -> F

D -> C

D -> E

E -> F

F -> C

F -> E

It can be represented by the following Python data structure:

graph = {'A': ['B', 'C'],

'B': ['C', 'D'],

'C': ['D',’F’],

'D': ['C',’E’],

'E': ['F'],

'F': ['C',’E’]}

This is a dictionary whose keys are the nodes of the graph. For each key, the corresponding value is a list containing the nodes that are connected by a direct arc from this node. This is about as simple as it gets (even simpler, the nodes could be represented by numbers instead of names, but names are more convenient and can easily be made to carry more information, such as city names).

Let's write a simple function to determine a path between two nodes. It takes a graph and the start and end nodes as arguments. It will return a list of nodes (including the start and end nodes) comprising the path. When no path can be found, it returns None. The same node will not occur more than once on the path returned (i.e. it won't contain cycles). The algorithm uses an important technique called *backtracking*: it tries each possibility in turn until it finds a solution.

def find\_path(graph, start, end, path=[]):

path = path + [start]

if start == end:

return path

if start not in graph:

return None

for node in graph[start]:

if node not in path:

newpath = find\_path(graph, node, end, path)

if newpath: return newpath

return None

A sample run of the function find\_path() (using the graph above):

>>> find\_path(graph, 'A', 'D')

['A', 'B', 'C', 'D']

**Example 2:**

######### Directed Weighted Graph and Search ############

class Graph:

def \_\_init\_\_(self, nodes=None, edges=None):

"""Initialize a graph object.

Args:

nodes: Iterator of nodes. Each node is an object.

edges: Iterator of edges. Each edge is a tuple of 2 nodes.

"""

self.nodes, self.adj = [], {}

if nodes != None:

self.add\_nodes\_from(nodes)

if edges != None:

self.add\_edges\_from(edges)

def \_\_len\_\_(self):

"""Returns the number of nodes in the graph.

>>> g = Graph(nodes=[x for x in range(7)])

>>> len(g)

7

"""

return len(self.nodes)

def \_\_contains\_\_(self, x):

"""Return true if a node x is in the graph.

>>> g = Graph(nodes=[x for x in range(7)])

>>> 6 in g

True

>>> 7 in g

False

"""

return x in self.nodes

def \_\_iter\_\_(self):

"""Iterate over the nodes in the graph.

>>> g = Graph(nodes=[x for x in range(7)])

>>> [x \* 2 for x in g]

[0, 2, 4, 6, 8, 10, 12]

"""

return iter(self.nodes)

def \_\_getitem\_\_(self, x):

"""Returns the iterator over the adjacent nodes of x.

>>> g = Graph(nodes=[x for x in range(7)], edges=[(1,0), (1,2), (1,3)])

>>> [x for x in g[1]]

[0, 2, 3]

"""

return iter(self.adj[x])

def \_\_str\_\_(self):

return 'V: %s\nE: %s' % (self.nodes, self.adj)

def add\_node(self, n):

if n not in self.nodes:

self.nodes.append(n)

self.adj[n] = []

def add\_nodes\_from(self, i):

for n in i:

self.add\_node(n)

def add\_edge(self, u, v): # undirected unweighted graph

self.adj[u] = self.adj.get(u, []) + [v]

self.adj[v] = self.adj.get(v, []) + [u]

def add\_edges\_from(self, i):

for n in i:

self.add\_edge(\*n)

def number\_of\_nodes(self):

return len(self.nodes)

def number\_of\_edges(self):

return sum(len(l) for \_, l in self.adj.items()) // 2

class DGraph(Graph):

def add\_edge(self, u, v):

self.adj[u] = self.adj.get(u, []) + [v]

class WGraph(Graph):

def \_\_init\_\_(self, nodes=None, edges=None):

"""Initialize a graph object.

Args:

nodes: Iterator of nodes. Each node is an object.

edges: Iterator of edges. Each edge is a tuple of 2 nodes and a weight.

"""

self.nodes, self.adj, self.weight = [], {}, {}

if nodes != None:

self.add\_nodes\_from(nodes)

if edges != None:

self.add\_edges\_from(edges)

def add\_edge(self, u, v, w):

self.adj[u] = self.adj.get(u, []) + [v]

self.adj[v] = self.adj.get(v, []) + [u]

self.weight[(u,v)] = w

self.weight[(v,u)] = w

def get\_weight(self, u, v):

return self.weight[(u,v)]

class DWGraph(WGraph):

def add\_edge(self, u, v, w):

self.adj[u] = self.adj.get(u, []) + [v]

self.weight[(u,v)] = w

###############################################################################

if \_\_name\_\_ == '\_\_main\_\_':

pass

**Lab Task:**

1. Change the function find\_path to return a list of all paths (without cycles) instead of the first path it finds.
2. Consider a simple (directed) graph (digraph) having six nodes (A-F) and the following arcs (directed edges) with respective cost of edge given in parentheses:

A -> B (2)

A -> C (1)

B -> C (2)

B -> D (5)

C -> D (1)

C -> F (3)

D -> C (1)

D -> E (4)

E -> F (3)

F -> C (1)

F -> E (2)

Using the code for a directed weighted graph in Example 2, instantiate an object of DWGraph in \_\_main\_\_, add the nodes and edges of the graph using the relevant functions, and implement a function find\_path() that takes starting and ending nodes as arguments and returns at least one path (if one exists) between those two nodes. The function should also keep track of the cost of the path and return the total cost as well as the path. Print the path and its cost in \_\_main\_\_.

Lab 5

**Uninformed Search:**

Recall your previous lab on Graphs in Python where you implemented a basic path finding algorithm for Directed, Weighted graphs.

**Depth-first search:**

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

**Pseudocode:**

Input: A graph G and a vertex v of G

Output: All vertices reachable from v labelled as discovered

A recursive implementation of DFS:

1 procedure DFS(G,v):

2 label v as discovered

3 for all edges from v to w in G.adjacentEdges(v) do

4 if vertex w is not labeled as discovered then

5 recursively call DFS(G,w)

A non-recursive implementation of DFS:

1 procedure DFS-iterative(G,v):

2 let S be a stack

3 S.push(v)

4 while S is not empty

5 v = S.pop()

6 if v is not labeled as discovered:

7 label v as discovered

8 for all edges from v to w in G.adjacentEdges(v) do

9 S.push(w)

**Iterative Deepening Depth-first search:**

Iterative Deepening Depth-first search (ID-DFS) is a state space/graph search strategy in which a depth-limited version of depth-first search is run repeatedly with increasing depth limits until the goal is found. IDDFS is equivalent to breadth-first search, but uses much less memory; on each iteration, it visits the nodes in the search tree in the same order as depth-first search, but the cumulative order in which nodes are first visited is effectively breadth-first.

**Pseudocode:**

The following pseudocode shows IDDFS implemented in terms of a recursive depth-limited DFS (called DLS).

**function** IDDFS(root)

**for** depth **from** 0 **to** ∞

found ← DLS(root, depth)

**if** found ≠ null

**return** found

**function** DLS(node, depth)

**if** depth = 0 **and** node is a goal

**return** node

**if** depth > 0

**foreach** child of node

found ← DLS(child, depth−1)

**if** found ≠ null

**return** found

**return** null

**Lab Task:**

Convert following maze into a state space tree and then apply backtracking using depth first search on the state space tree. Start state is A and goal state is T

**NOTE**: Avoid cycles/ loops while making state space tree. Solid line in maze shows the wall and dotted line shows the open path. Only horizontal and vertical movements are allowed.

Q R S T

M N O P

I J K L

E F G H

A B C D

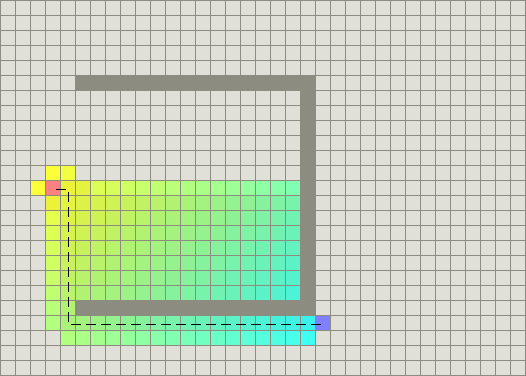
Lab 6

**Searching:**

Some computational problems can have many solutions. We say that these solutions are lying in the search space. Out of the many possible solutions, we try to use a searching algorithm that finds the optimal solution present in the search space.

**A\* Search:**

A\* Search Algorithm was developed in 1968 to combine heuristic approaches like Greedy Best-First-Search and formal approaches like Dijsktra’s algorithm. It’s a little unusual in that heuristic approaches usually give you an approximate way to solve problems without guaranteeing that you get the best answer. However, A\* is built on top of the heuristic, and although the heuristic itself does not give you a guarantee, A\* *can* guarantee a shortest path. A\* is like Dijkstra’s algorithm in that it can be used to find a shortest path. A\* is like Greedy Best-First-Search in that it can use a heuristic to guide itself. In the simple case, it is as fast as Greedy Best-First-Search.



The secret to its success is that it combines the pieces of information that Dijkstra’s algorithm uses (favouring vertices that are close to the starting point) *and* information that Greedy Best-First-Search uses (favouring vertices that are close to the goal). In the standard terminology used when talking about A\*, g(n) represents the *exact cost* of the path from the starting point to any vertex n, and h(n) represents the heuristic *estimated cost* from vertex n to the goal.

In the above diagram, the yellow (h) represents vertices far from the goal and teal (g) represents vertices far from the starting point. A\* balances the two as it moves from the starting point to the goal. Each time through the main loop, it examines the vertex n that has the lowest f(n) = g(n) + h(n).

The heuristic can be used to control A\*’s behaviour.

* At one extreme, if h(n) is 0, then only g(n) plays a role, and A\* turns into Dijkstra’s algorithm, which is guaranteed to find a shortest path.
* If h(n) is always lower than (or equal to) the cost of moving from n to the goal, then A\* is guaranteed to find a shortest path. The lower h(n) is, the more node A\* expands, making it slower.
* If h(n) is exactly equal to the cost of moving from n to the goal, then A\* will only follow the best path and never expand anything else, making it very fast. Although you can’t make this happen in all cases, you can make it exact in some special cases. It’s nice to know that given perfect information, A\* will behave perfectly.
* If h(n) is sometimes greater than the cost of moving from n to the goal, then A\* is not guaranteed to find a shortest path, but it can run faster.
* At the other extreme, if h(n) is very high relative to g(n), then only h(n) plays a role, and A\* turns into Greedy Best-First-Search.

So we have an interesting situation in that we can decide what we want to get out of A\*. At exactly the right point, we’ll get shortest paths really quickly. If we’re too low, then we’ll continue to get shortest paths, but it’ll slow down. If we’re too high, then we give up shortest paths, but A\* will run faster.

**Importance of Scale:**

A\* computes f(n) = g(n) + h(n). To add two values, those two values need to be at the same scale. If g(n) is measured in hours and h(n) is measured in meters, then A\* is going to consider g or h too much or too little, and you either won’t get as good paths or you A\* will run slower than it could.

**Algorithm:**

The A\* algorithm, stripped of all the code, is fairly simple. There are two sets, OPEN and CLOSED. The OPEN set contains those nodes that are candidates for examining. Initially, the OPEN set contains only one element: the starting position. The CLOSED set contains those nodes that have already been examined. Initially, the CLOSED set is empty. Graphically, the OPEN set is the “frontier” and the CLOSED set is the “interior” of the visited areas. Each node also keeps a pointer to its parent node so that we can determine how it was found.

There is a main loop that repeatedly pulls out the best node n in OPEN (the node with the lowest f value) and examines it. If n is the goal, then we’re done. Otherwise, node n is removed from OPEN and added to CLOSED. Then, its neighbours n′ are examined. A neighbour that is in CLOSED has already been seen, so we don’t need to look at it. You do need to check to see if the node’s g value can be lowered, and if so, you re-open it. A neighbour that is in OPEN is scheduled to be looked at, so we don’t need to look at it now. Otherwise, we add it to OPEN, with its parent set to n. The path cost to n′, g(n′), will be set to g(n) + movementcost(n, n′).

OPEN = priority queue containing START

CLOSED = empty set

while lowest rank in OPEN is not the GOAL:

current = remove lowest rank item from OPEN

add current to CLOSED

for neighbours of current:

cost = g(current) + movementcost(current, neighbour)

if neighbour in OPEN and cost less than g(neighbour):

remove neighbour from OPEN, because new path is better

if neighbour in CLOSED and cost less than g(neighbour):

remove neighbour from CLOSED

if neighbour not in OPEN and neighbour not in CLOSED:

set g(neighbour) to cost

add neighbour to OPEN

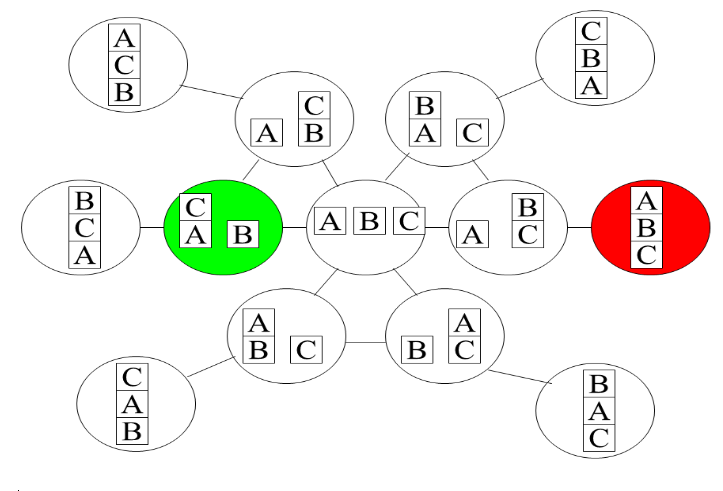
set priority queue rank to g(neighbour) + h(neighbour)

set neighbour's parent to current

reconstruct reverse path from goal to start by following parent pointers

**Lab Task:**

You need to arrange three boxes labeled as A, B and C. State space graph of the problem is shown below.



Start state is and Goal state is . Apply A\* search to reach the goal. Calculate total cost for goal.

B

C

A

A

B

C

h(n) = “# of blocks misplaced (if the upper block is not at its goal position”

g(n) = “# of steps so far”

Lab 7

**Searching:**

* Some computational problems can have many solutions. We say that these solutions are lying in the search space.
* The idea is that out of the many possible solutions, we use a searching algorithm that finds the optimal solution present in the search space.
* The solution found can be optimal locally or globally, but that depends on the type of search that has been implemented.

**Local Search:**

* Local search can be used on problems where a solution must be found that maximizes a criterion among a number of candidate solutions.
* A local search algorithm starts from a candidate solution and then iteratively moves to a neighbour solution.

**Games vs. search problems**:

* "Unpredictable" opponent → Solution strategy should be to specify a move for every possible opponent reply.
* Time limits → if time limits are in place it becomes unlikely to find goal as performing search for every possible move the opponent makes is expensive. In such a situation, the algorithm must approximate.

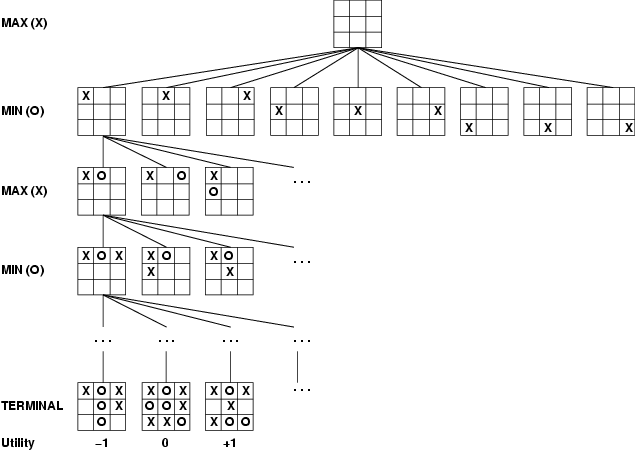
**Optimal Decision in Games:**

* The initial state: It identifies the player to move and the board position
* A successor function: It returns a list of (move, state) pairs, each indicating a legal move and the resulting state.
* A terminal test: determines when the game is over. States where the game ends are called terminal states.
* A utility function: Gives a numerical value to the terminal states. For example the outcome is a Win (+1), a loss (-1) or a draw (0).

**Game Tree:**

* It is a tree where you define moves for every possible move/reply of the opponent
* Your aim is to find the route that gives the terminal state as a **win**

**Game tree for tic-tac-toe(2-player, deterministic, turns):**



**Minimax Algorithm:**

## Minimax and Alpha-Beta Types and Functions

Any implementation of minimax and alpha-beta must be supplied with the following types and routines.

**board type**

This type contains all information specific to the current state of the game, including layout of the board and current player.

**score type**

This data type indicates piece advantage, strategic advantage, and possible wins. In most games, strategic advantage includes the number of moves available to each player with the goal of minimizing the opponent's mobility.

**neg\_infinity** and **pos\_infinity**

The most extreme scores possible in the game, each most disadvantageous for one player in the game.

**generate\_moves**

This function takes the current board and generates a list of possible moves for the current player.

**apply\_move**

This function takes a board and a move, returning the board with all the updates required by the given move.

**null\_move**

If the chosen game allows or requires a player to forfeit moves in the case where no moves are available, this function takes the current board and returns it, after switching the current player.

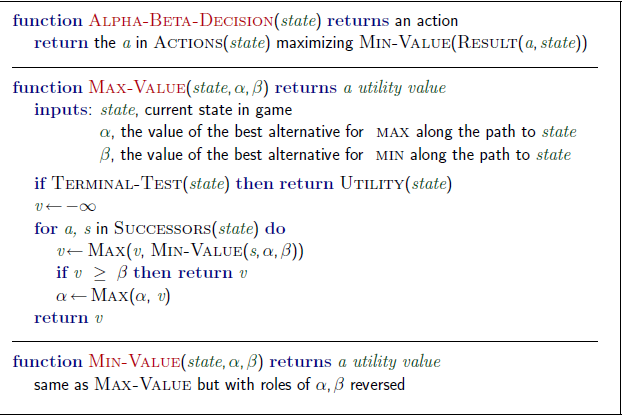
**static\_evaluation**

This function takes the board as input and returns a score for the game.

**compare\_scores**

This function takes 2 scores to compare and a player, returning the score that is more advantageous for the given player. If scores are stored as simple integers, this function can be the standard < and > operators.

**Minimax Pseudocode**



**Pseudocode:**

alpaBetaMinimax(node, alpha, beta)

"""

Returns best score for the player associated with the given node.

Also sets the variable bestMove to the move associated with the

best score at the root node.

"""

# check if at search bound

if node is at depthLimit

return staticEval(node)

# check if leaf

children = successors(node)

if len(children) == 0

if node is root

bestMove = []

return staticEval(node)

# initialize bestMove

if node is root

bestMove = operator of first child

# check if there is only one option

if len(children) == 1

return None

if it is MAX's turn to move

for child in children

result = alphaBetaMinimax(child, alpha, beta)

if result > alpha

alpha = result

if node is root

bestMove = operator of child

if alpha >= beta

return alpha

return alpha

if it is MIN's turn to move

for child in children

result = alphaBetaMinimax(child, alpha, beta)

if result < beta

beta = result

if node is root

bestMove = operator of child

if beta <= alpha

return beta

return beta

**Standard Implementation of Minimax:**

The standard implementation of the Minimax algorithm frequently includes three functions: minimax(game\_state), min\_play(game\_state) and max\_play(game\_state). Note that Python is used here as working pseudo-code.

**def** **minimax**(game\_state):

moves **=** game\_state**.**get\_available\_moves()

best\_move **=** moves[0]

best\_score **=** float('-inf')

**for** move **in** moves:

clone **=** game\_state**.**next\_state(move)

score **=** min\_play(clone)

**if** score **>** best\_score:

best\_move **=** move

best\_score **=** score

**return** best\_move

To summarize, Minimax is given a game state, obtains a set of valid moves from the game state, simulates all valid moves on clones of the game state, evaluates each game state which follows a valid move and finally returns the best move.

The following two helper functions simulate play between both the opposing player and the current player through the min\_play and max\_play procedures respectively. With the aid of these two helper functions, the entire game tree is traversed recursively given the current state of the game.

**def** **min\_play**(game\_state):

**if** game\_state**.**is\_gameover():

**return** evaluate(game\_state)

moves **=** game\_state**.**get\_available\_moves()

best\_score **=** float('inf')

**for** move **in** moves:

clone **=** game\_state**.**next\_state(move)

score **=** max\_play(clone)

**if** score **<** best\_score:

best\_move **=** move

best\_score **=** score

**return** best\_score

**def** **max\_play**(game\_state):

**if** game\_state**.**is\_gameover():

**return** evaluate(game\_state)

moves **=** game\_state**.**get\_available\_moves()

best\_score **=** float('-inf')

**for** move **in** moves:

clone **=** game\_state**.**next\_state(move)

score **=** min\_play(clone)

**if** score **>** best\_score:

best\_move **=** move

best\_score **=** score

**return** best\_score

In particular, the opponent intends to minimize the current player's score and the current player intends to maximize their own score. Note that the helper functions short-circuit and return early if the game is over.

**Tic Tac Toe:**

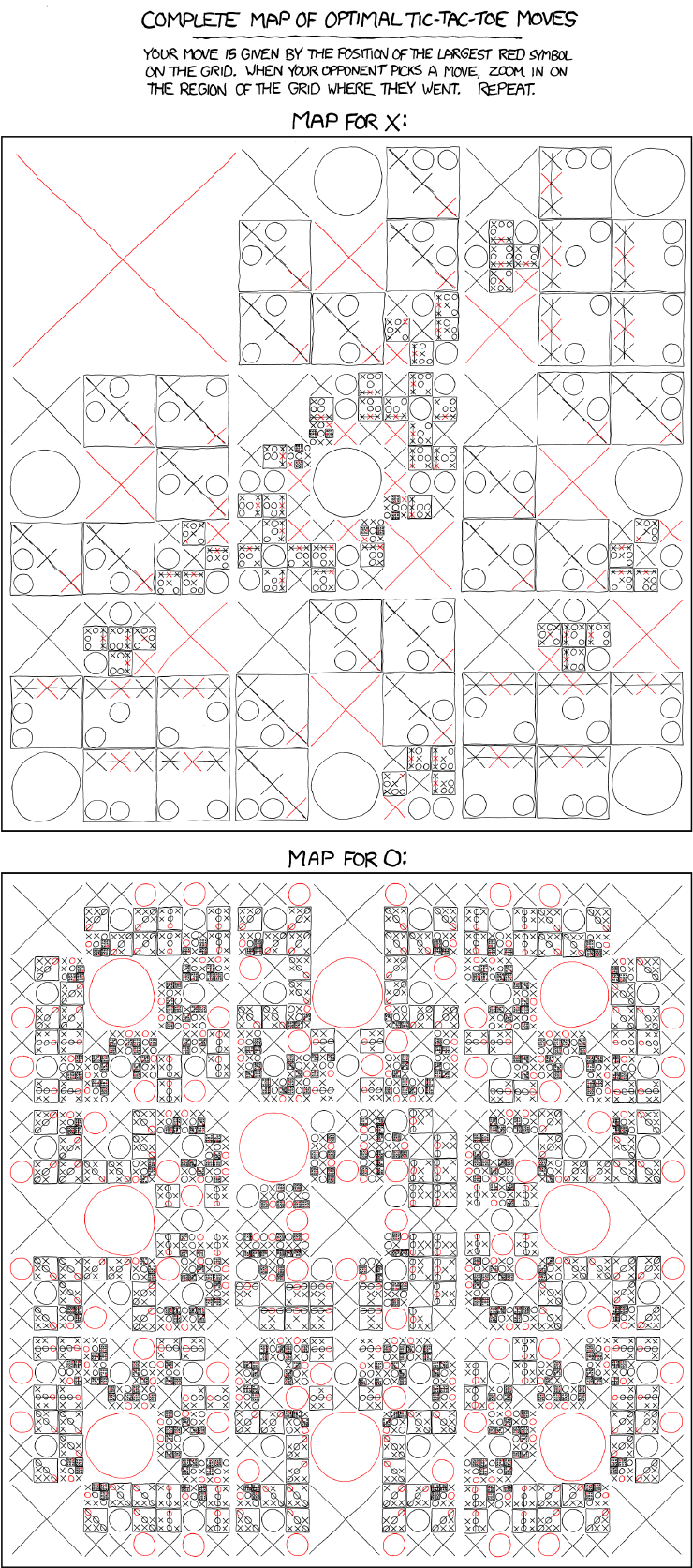
Tic Tac Toe is a famous two player, zero-sum game which can illustrate the use of Game Search algorithms such as minimax very well as each player tries his best to win and to make a move that minimizes the opponent’s chances of winning.

The complete map for optimal Tic Tac Toe moves is given on the next page. (credits: https://xkcd.com/832/)

**Lab Task:**

Implement Tic Tac Toe or Minesweeper using Minimax alpha-beta pruning algorithm. You may use one of the pseudocodes provided in this Lab manual.

You might need to install numPy. Please refer to the document “Installing Python packages using pip3.pdf”



Lab 8

**Constraint Satisfaction Problems:**

*The more constraints one imposes, the more one frees oneself of the chains that shackle the spirit.* – Igor Stravinsky, *Poetics of Music*

The formalism of search problems we have discussed (in Labs 4 and 7) so far is a very powerful formalism that depends on the notion of **state**. From the point of view of a search algorithm, a state is a black box with no discernible internal structure. A state can be represented by an arbitrary data structure and can be accessed only by **problem speciﬁc** functions: successor, goal test, heuristics etc.

The framework used in this lab (constraint satisfaction problems) admits a very simple standard representation. This allows us to deﬁne search algorithms that **take advantage** of this very **simple** representation and **use general purpose heuristics** to enable solution of large problems.

The simple structure also allows us to deﬁne methods for problem decomposition and oﬀers us an intimate connection between the structure of a problem and the diﬃculty of solving it.

**CSP Definitions:**

A constraint satisfaction problem (CSP) is deﬁned by:

* A set of **variables** X1,...,Xn. Each variable has a domain D*i* of possible values.
* A set of **constraints** C1,...,Cm. Each constraint involves some subset of the variables and speciﬁes the allowable combinations of values for that subset.

Formally, an (k-ary) **constraint** C on a set of variables X1,...,Xk is a subset of the Cartesian product D1 ×···×Dk.

* A **solution** to a CSP is a complete assignment of values to variables such that all the constraints are satisﬁed.
* A CSP is called **consistent** if it has a solution, otherwise it is called **inconsistent**.

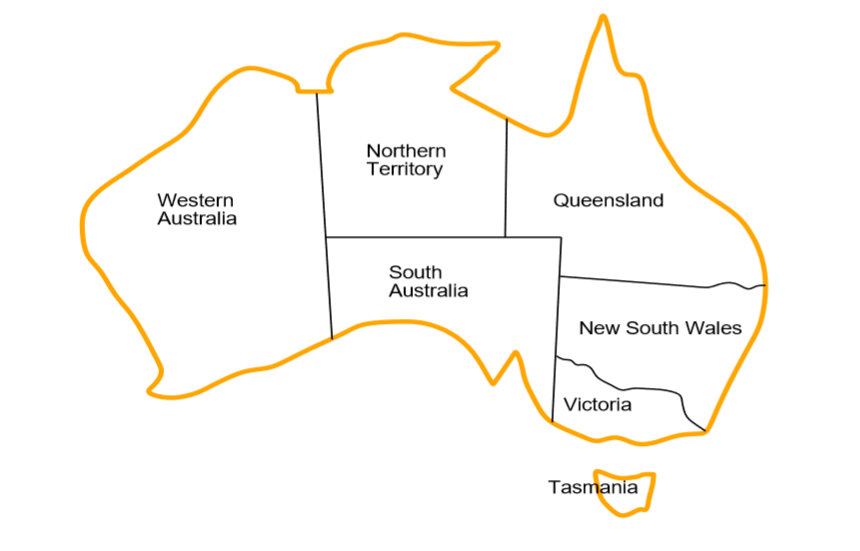
**Example:**

Map colouring problem, N-Puzzle, N-Queens problem, etc.

Let us look at some of these constraint satisfaction problems to understand them better.

1. **Map Colouring Problem**

The goal is to assign colours to each region so that no neighbouring regions have the same colour.



**Formal Definition:**

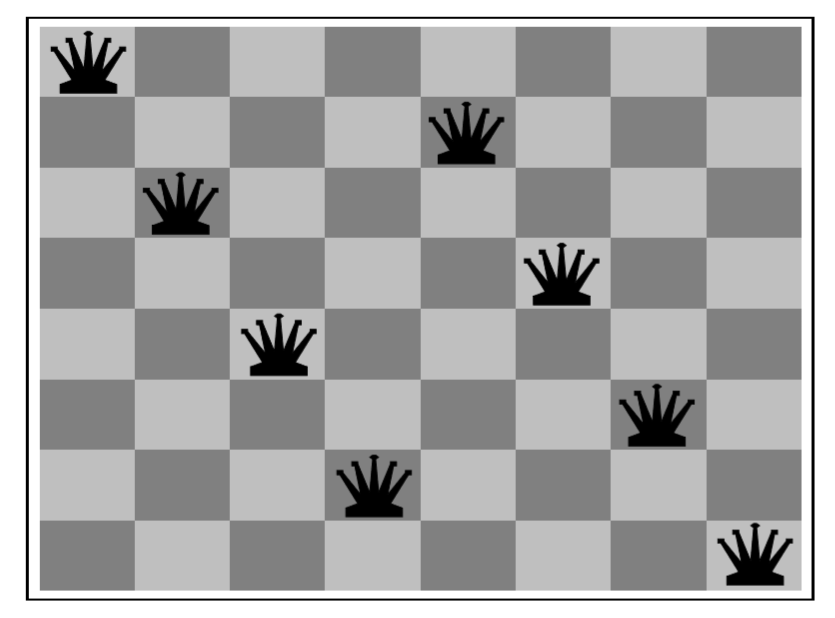
* Variables: WA,NT,SA,Q,NSW,V,T
* Domain (same for all variables): { red, green, blue }
* Constraints:

C(WA,NT) ={ (red,green), (red,blue), (green,red), (blue,red), (blue,green) , (green, blue)}

More succinctly, WA!= NT. Similarly for the other pairs of variables.

1. **8-Queens problem**

The goal is to place each queen in a square so that no queen attacks any other queen. (In chess, a queen can attack another chess piece within the same column, row, or diagonal.)



**Formal Definition:**

• Variables: Let variable Xi (i = 1,...,8) represent the column that the i-th queen occupies in the i-th row. If columns are represented by numbers 1 to 8 then the domain of every variable Xi is Di ={1,2,...,8}.

• Constraints: There is a binary constraint C(Xi,Xj) for each pair of variables. These constraints can be speciﬁed succinctly as follows:

* For all variables Xi and Xj, Xi != Xj.
* For all variables Xi and Xj, if Xi = a and Xj = b then i−j != a−b and i−j != b−a.

**CSPs and their solutions:**

A CSP can be given an incremental formulation as a standard search problem as follows:

* Initial state: the empty assignment {}, in which all variables are unassigned.
* Successor function: a value can be assigned to any unassigned variable, if it does not conﬂict with previously assigned variables.
* Goal test: the current assignment is complete.
* Path cost: a constant cost (e.g., 1) for every step.

Every solution must be a complete assignment and therefore appears at depth n if there are n variables. Furthermore, the search tree extends only to depth n. For these reasons, depth-ﬁrst search algorithms are popular for CSPs. It is also the case that *the path by which a solution is reached is irrelevant*. Hence, we can also use a **complete-state formulation**, in which every state is a complete assignment that might or might not satisfy the constraints. Local search methods work well for this formulation.

The simplest kind of CSP involves variables that are discrete and have **ﬁnite domains**. Map-colouring problems are of this kind. The 8-queens problem can also be viewed as a ﬁnite-domain CSP, where the variables Q1,...,Q8 are the positions of each queen in columns 1,...,8 and each variable has the domain {1,2,3,4,5,6,7,8}. If the maximum domain size of any variable in a CSP is d, then the number of possible complete assignments is O(dn)—that is, exponential in the number of variables. Finite-domain CSPs include **Boolean CSPs**, whose variables can be either *true* or *false*. Boolean CSPs include as special cases some NP-complete problems. In the worst case, therefore, we cannot expect to solve ﬁnite-domain CSPs in less than exponential time. In most practical applications, however, **general-purpose CSP algorithms** can solve problems orders of magnitude larger than those solvable via the general-purpose search algorithms.

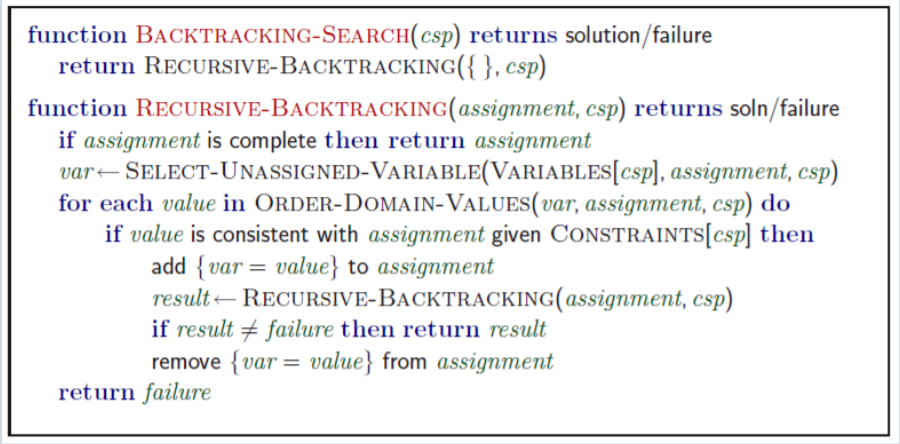
In addition to examining the types of variables that can appear in CSPs, it is useful to look at the types of constraints. The simplest type is the **unary constraint**, which restricts the value of a single variable. For example, it could be the case that South Australians actively dislike the colour green. Every unary constraint can be eliminated simply by pre-processing the domain of the corresponding variable to remove any value that violates the constraint. A **binary constraint** relates two variables. For example, SA != NSW is a binary constraint. A binary CSP is one with only binary constraints; it can be represented as a constraint graph.

**Solving a CSP:**

CSPs can be solved in several ways. The simplest way is to use backtracking which is depth first search with single variable assignment.

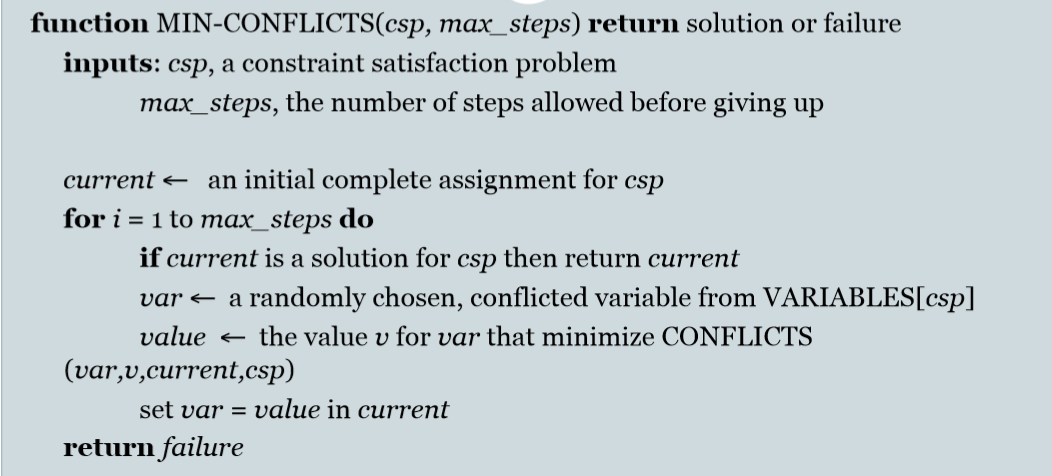
**Backtracking:**

* Variable assignments are commutative, i.e., [WA=red then NT =green] same as [NT =green then WA=red]
* Only need to consider assignments to a single variable at each node ➩ b=d and there are dn leaves
* Backtracking search is the basic uninformed algorithm for CSPs
* Backtracking=depth-first search with one variable assigned per node.



**Local Search for CSPs:**

* Use complete-state representation
  + Initial state = all variables assigned values
  + Successor states = change 1 (or more) values
* For CSPs
  + allow states with unsatisfied constraints (unlike backtracking)
  + operators reassign variable values
  + hill-climbing with n-queens is an example
* Variable selection: randomly select any conflicted variable.
* Value selection: **min-conflicts heuristic**
  + Select new value that results in a minimum number of conflicts with the other variables



**Analysis Lab Tasks:**

1. Apply forward checking to solve the problem of hotel room assignment. Map of rooms is given below. Two adjacent rooms should not be allocated to persons of same country. Persons x,y and z belongs to country P. Persons a and b belongs to country Q, person s belong to country R.

A

E

B

F

C

D

1. In an exhibition, stalls need to get allocated. The allocation should be according to following conditions.

* There are total three blocks (A, B, C).
* There are six types of stalls. (food, cosmetics, garments, toys, shoes, jewellery).
* Domain(A)= (food,toys,shoes,jewellery)
* Domain(B)= (toys, shoes, jewellery).
* Domain(C)= (food, cosmetics, garments, toys, shoes, jewellery).
* Block A can have only those types of stalls which are in Block C.
* Block C can have only those types of stalls which are in Block A.
* There should be no stall type in Block A that exists in Block B.
* Block B can have maximum of two types of stalls that are common to stalls in block C.

Lab 9

**Tkinter:**

The Tkinter module that comes with Python is as close as Python comes to having an official graphics user interface (GUI), but no documentation is installed.

**Our First Tkinter Program (File: hello1.py)**

from tkinter import \* #Please note small case lettering for tkinter

root = Tk()

w = Label(root, text="Hello, world!")

w.pack()

root.mainloop()

**Running the Example**

To run the program, run the script as usual. The following window appears.

http://effbot.org/media/cache/3fd3019d8414d19ed155451fd9e4902e.gif

To stop the program, just close the window.

**Details**

We start by importing the tkinter module. It contains all classes, functions and other things needed to work with the Tk toolkit. In most cases, you can simply import everything from **Tkinter** into your module’s namespace:

from tkinter import \*

To initialize Tkinter, we have to create a Tk **root** widget. This is an ordinary window, with a title bar and other decoration provided by your window manager. You should only create one root widget for each program, and it must be created before any other widgets. The Tk class is instantiated without arguments. This creates a toplevel widget of Tk which usually is the main window of an application.

root = Tk()

Next, we create a **Label** widget as a child to the root window:

w = Label(root, text="Hello, world!")

w.pack()

A **Label** widget can display either text or an icon or other image. In this case, we use the **text** option to specify which text to display.

Next, we call the **pack** method on this widget. This tells it to size itself to fit the given text, and make itself visible. However, the window won’t appear until we’ve entered the Tkinter event loop:

root.mainloop()

The program will stay in the event loop until we close the window. The event loop doesn’t only handle events from the user (such as mouse clicks and key presses) or the windowing system (such as redraw events and window configuration messages), it also handles operations queued by Tkinter itself. Among these operations are geometry management (queued by the **pack** method) and display updates. This also means that the application window will not appear before you enter the main loop.

**Our Second Tkinter Program**

from tkinter import \*

class App:

def \_\_init\_\_(self, master):

frame = Frame(master)

frame.pack()

self.button = Button(

frame, text="QUIT", fg="red", command=frame.quit

)

self.button.pack(side=LEFT)

self.hi\_there = Button(frame, text="Hello", command=self.say\_hi)

self.hi\_there.pack(side=LEFT)

def say\_hi(self):

print "hi there, everyone!"

root = Tk()

app = App(root)

root.mainloop()

root.destroy() # optional; see description below

**Running the Example**

When you run this example, the following window appears.

http://effbot.org/media/cache/76734d5c44455920d98197bd2958fec7.gif

This sample application is written as a class. The constructor (the **\_\_init\_\_** method) is called with a parent widget (the **master**), to which it adds a number of child widgets. The constructor starts by creating a **Frame** widget. A frame is a simple container, and is in this case only used to hold the other two widgets.

class App:

def \_\_init\_\_(self, master):

frame = Frame(master)

frame.pack()

The frame instance is stored in a local variable called **frame**. After creating the widget, we immediately call the **pack** method to make the frame visible.

We then create two **Button** widgets, as children to the frame.

self.button = Button(frame, text="QUIT", fg="red", command=frame.quit)

self.button.pack(side=LEFT)

self.hi\_there = Button(frame, text="Hello", command=self.say\_hi)

self.hi\_there.pack(side=LEFT)

This time, we pass a number of **options** to the constructor, as keyword arguments. The first button is labelled “QUIT”, and is made red (**fg** is short for **foreground**). The second is labelled “Hello”. Both buttons also take a **command** option. This option specifies a function, or (as in this case) a bound method, which will be called when the button is clicked.

The button instances are stored in instance attributes. They are both packed, but this time with the **side=LEFT** argument. This means that they will be placed as far left as possible in the frame; the first button is placed at the frame’s left edge, and the second is placed just to the right of the first one (at the left edge of the *remaining space* in the frame, that is). By default, widgets are packed relative to their parent (which is **master** for the frame widget, and the frame itself for the buttons). If the side is not given, it defaults to **TOP**.

The “hello” button callback is given next. It simply prints a message to the console every time the button is pressed:

def say\_hi(self):

print "hi there, everyone!"

Finally, we provide some script level code that creates a **Tk** root widget, and one instance of the **App** class using the root widget as its parent:

root = Tk()

app = App(root)

root.mainloop()

root.destroy()

The **mainloop** call enters the Tk event loop, in which the application will stay until the **quit** method is called (just click the QUIT button), or the window is closed.

The **destroy** call is only required if you run this example under certain development environments; it explicitly destroys the main window when the event loop is terminated. Some development environments won’t terminate the Python process unless this is done.

More information on callback functions and how to associate them with a particular widget e.g. a button:

To use a function object as a callback, pass it directly to Tkinter.

from tkinter import \*

def **callback**():

print ("clicked!")

b = Button(text="click me", command=callback)

b.pack()

mainloop()

For each function object, the Tkinter interface layer registers a Tk command with a unique name. When that Tk command is called by the Button implementation, the command calls the corresponding Python function.

### Passing Argument to Callbacks

Tkinter’s Button widget doesn’t pass any information to the callback. This makes things a bit complicated if you want to use the same callback for several buttons, like in this example:

def **callback**():

print ("button", "?")

Button(text="one", command=callback).pack()

Button(text="two", command=callback).pack()

Button(text="three", command=callback).pack()

A common beginner’s mistake is to *call* the callback function when constructing the widget. That is, instead of giving just the function’s name (e.g. “callback”), the programmer adds parentheses and argument values to the function:

def **callback**(number):

print ("button", number)

Button(text="one", command=callback(1)).pack()

Button(text="two", command=callback(2)).pack()

Button(text="three", command=callback(3)).pack()

If you do this, Python will call the callback function *before* creating the widget, and pass the function’s *return value* to Tkinter. Tkinter then attempts to convert the return value to a string, and tells Tk to call a function with that name when the button is activated. This is probably not what you wanted.

For simple cases like this, you can use a **lambda** expression as a link between Tkinter and the callback function:

def **callback**(number):

print ("button", number)

Button(text="one", command=lambda: callback(1)).pack()

Button(text="two", command=lambda: callback(2)).pack()

Button(text="three", command=lambda: callback(3)).pack()

# The Tkinter Entry Widget

The **Entry** widget is a standard Tkinter widget used to enter or display a single line of text. To create an Entry field you can use for typing/entering data:

my\_box = Entry(frame, justify=RIGHT)

my\_box.grid(row = 0, column = 0, columnspan = 24, padx = 2, pady = 5)

## When to use the Entry Widget

The entry widget is used to enter text strings. This widget allows the user to enter one line of text, in a single font.

To enter multiple lines of text, use the [**Text**](http://effbot.org/tkinterbook/text.htm) widget.

## Patterns

To add entry text to the widget, use the **insert** method. To replace the current text, you can call **delete** before you insert the new text.

e = Entry(master)

e.pack()

e.delete(0, END)

e.insert(0, "a default value")

To fetch the current entry text, use the **get** method:

s = e.get()

You can also bind the entry widget to a **StringVar** instance, and set or get the entry text via that variable:

v = StringVar()

e = Entry(master, textvariable=v)

e.pack()

v.set("a default value")

s = v.get()

This example creates an Entry widget, and a Button that prints the current contents:

from tkinter import \*

master = Tk()

e = Entry(master)

e.pack()

e.focus\_set()

def **callback**():

print (e.get())

b = Button(master, text="get", width=10, command=callback)

b.pack()

mainloop()

## Concepts

### Indexes

The *Entry* widget allows you to specify character positions in a number of ways:

* Numerical indexes
* **ANCHOR**
* **END**
* **INSERT**
* Mouse coordinates (“@x”)

**Numerical indexes** work just like Python list indexes. The characters in the string are numbered from 0 and upwards. You specify ranges just like you slice lists in Python: for example, (0, 5) corresponds to the first five characters in the entry widget.

**ANCHOR** (or the string “anchor”) corresponds to the start of the selection, if any. You can use the **select\_from** method to change this from the program.

**END** (or “end”) corresponds to the position just after the last character in the entry widget. The range (0, END) corresponds to all characters in the widget.

**INSERT** (or “insert”) corresponds to the current position of the text cursor. You can use the **icursor** method to change this from the program.

Finally, you can use the mouse position for the index, using the following syntax:

"@%d" % x

where *x* is given in pixels relative to the left edge of the entry widget.

## Reference

**Entry(master=None, \*\*options)** (class)

A text entry field.

*master* : Parent widget.

*\*\*options* : Widget options. See the description of the [**config**](http://effbot.org/tkinterbook/entry.htm#entry.Entry.config-method) method for a list of available options.

**config(\*\*options)**

Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

*\*\*options* :Widget options.

*background=*

Widget background. The default is system specific. (the option database name is background, the class is Background)

*bg=*

Same as **background**.

*borderwidth=*

Border width. The default is system specific, but is usually a few pixels. (borderWidth/BorderWidth)

*bd=*

Same as **borderwidth**.

*cursor=*

Widget cursor. The default is a text insertion cursor (typically an “I-beam” cursor, e.g. **xterm**). (cursor/Cursor)

*disabledbackground=*

Background to use when the widget is disabled. If omitted or blank, the standard background is used instead. (disabledBackground/DisabledBackground)

*disabledforeground=*

Text color to use when the widget is disabled. If omitted or blank, the standard foreground is used instead. (disabledForeground/DisabledForeground)

*exportselection=*

If true, selected text is automatically exported to the clipboard. Default is true. (exportSelection/ExportSelection)

*font=*

Widget font. The default is system specific. (font/Font)

*foreground=*

Text color. (foreground/Foreground)

*fg=*

Same as **foreground**.

*highlightbackground=*

Together with **highlightcolor**, this option controls how to draw the focus highlight border. This option is used when the widget doesn’t have focus. The default is system specific. (highlightBackground/HighlightBackground)

*highlightcolor=*

Same as **highlightbackground**, but is used when the widget has focus. (highlightColor/HighlightColor)

*highlightthickness=*

The width of the focus highlight border. Default is typically a few pixels, unless the system indicates focus by modifying the button itself (like on Windows). (highlightThickness/HighlightThickness)

*insertbackground=*

Color used for the insertion cursor. (insertBackground/Foreground)

*insertborderwidth=*

Width of the insertion cursor’s border. If this is set to a non-zero value, the cursor is drawn using the **RAISED** border style. (insertBorderWidth/BorderWidth)

*insertofftime=*

Together with **insertontime**, this option controls cursor blinking. Both values are given in milliseconds. (insertOffTime/OffTime)

*insertontime=*

See **insertofftime**. (insertOnTime/OnTime)

*insertwidth=*

Width of the insertion cursor. Usually one or two pixels. (insertWidth/InsertWidth)

*invalidcommand=*

FIXME. No default. (invalidCommand/InvalidCommand)

*invcmd=*

Same as **invalidcommand**.

*justify=*

How to align the text inside the entry field. Use one of **LEFT**,**CENTER**, or **RIGHT**. The default is **LEFT**. (justify/Justify)

*readonlybackground=*

The background color to use when the state is “readonly”. If omitted or blank, the standard background is used instead. (readonlyBackground/ReadonlyBackground)

*relief=*

Border style. The default is **SUNKEN**. Other possible values are **FLAT**, **RAISED**, **GROOVE**, and **RIDGE**.

*selectbackground=*

Selection background color. The default is system and display specific. (selectBackground/Foreground)

*selectborderwidth=*

Selection border width. The default is system specific. (selectBorderWidth/BorderWidth)te

*selectforeground=*

Selection text color. The default is system specific. (selectForeground/Background)

*show=*

Controls how to display the contents of the widget. If non-empty, the widget displays a string of characters instead of the actual contents. To get a password entry widget, set this option to “\*”. (show/Show)

*state=*

The entry state: **NORMAL**, **DISABLED**, or “readonly” (same as **DISABLED**, but contents can still be selected and copied). Default is **NORMAL**. Note that if you set this to**DISABLED** or “readonly”, calls to [**insert**](http://effbot.org/tkinterbook/entry.htm#entry.Entry.insert-method) and [**delete**](http://effbot.org/tkinterbook/entry.htm#entry.Entry.delete-method) are ignored. (state/State)

*takefocus=*

Indicates that the user can use the **Tab** key to move to this widget. Default is an empty string, which means that the entry widget accepts focus only if it has any keyboard bindings (default is on, in other words). (takeFocus/TakeFocus)

*textvariable=*

Associates a Tkinter variable (usually a **StringVar**) to the contents of the entry field. (textVariable/Variable)

*validate=*

Specifies when validation should be done. You can use “focus” to validate whenever the widget gets or loses the focus, “focusin” to validate only when it gets focus, “focusout” to validate when it loses focus, “key” on any modification, and ALL for all situations. Default is **NONE** (no validation). (validate/Validate)

*validatecommand=*

A function or method to call to check if the contents is valid. The function should return a true value if the new contents is valid, or false if it isn’t. Note that this option is only used if the**validate** option is not **NONE**. (validateCommand/ValidateCommand)

*vcmd=*

Same as **validatecommand**.

*width=*

Width of the entry field, in character units. Note that this controlS the size on screen; it does not limit the number of characters that can be typed into the entry field. The default width is 20 character. (width/Width)

*xscrollcommand=*

Used to connect an entry field to a horizontal scrollbar. This option should be set to the **set** method of the corresponding scrollbar. (xScrollCommand/ScrollCommand)

**delete(first, last=None)**

Deletes the character at index, or within the given range. Use delete(0, END) to delete all text in the widget.

*first* : Start of range.

*last* : Optional end of range. If omitted, only a single character is removed.

**get()**

Gets the current contents of the entry field.

Returns: The widget contents, as a string.

**icursor(index)**

Moves the insertion cursor to the given index. This also sets the **INSERT** index.

*index* :Where to move the cursor.

**index(index)**

Gets the numerical position corresponding to the given index.

*index* :An index.

Returns: The corresponding numerical index.

**insert(index, string)**

Inserts text at the given index. Use insert(INSERT, text) to insert text at the cursor, insert(END, text) to append text to the widget.

*index* :Where to insert the text.

*string :*The text to insert.

**scan\_dragto(x)**

Sets the scanning anchor for fast horizontal scrolling to the given mouse coordinate.

*x* :Current horizontal mouse position.

**scan\_mark(x)**

Scrolls the widget contents sideways according to the given mouse coordinate. The text is moved 10 times the distance between the scanning anchor and the new position.

*x* :Current horizontal mouse position.

**select\_adjust(index) :** Same as [**selection\_adjust**](http://effbot.org/tkinterbook/entry.htm#entry.Entry.selection_adjust-method).

**select\_clear() :** Same as [**selection\_clear**](http://effbot.org/tkinterbook/entry.htm#entry.Entry.selection_clear-method).

**select\_from(index) :** Same as [**selection\_from**](http://effbot.org/tkinterbook/entry.htm#entry.Entry.selection_from-method).

**select\_present() :** Same as [**selection\_present**](http://effbot.org/tkinterbook/entry.htm#entry.Entry.selection_present-method).

**select\_range(start, end) :** Same as [**selection\_range**](http://effbot.org/tkinterbook/entry.htm#entry.Entry.selection_range-method).

**select\_to(index) :** Same as [**selection\_to**](http://effbot.org/tkinterbook/entry.htm#entry.Entry.selection_to-method).

**selection\_adjust(index) :** Adjusts the selection to include also the given character. If index is already selected, do nothing.

*index* :The index.

**selection\_clear() :** Clears the selection.

**selection\_from(index) :** Starts a new selection. This also sets the **ANCHOR** index.

*index* :The index.

**selection\_present() :** Checks if text is selected. Returns: A true value if some part of the text is selected.

**selection\_range(start, end) [**[**#**](http://effbot.org/tkinterbook/entry.htm#Tkinter.Entry.selection_range-method)**]**

Explicitly sets the selection range. Start must be smaller than end. Use**selection\_range(0, END)** to select all text in the widget.

*Start* : Start of selection range.

*end* :End of range.

**selection\_to(index) :** Selects all text between **ANCHOR** and the given index.

**xview(index) :** Makes sure the given index is visible. The entry view is scrolled if necessary.

**xview\_moveto(fraction) :** Adjusts the entry view so that the given offset is at the left edge of the canvas. Offset 0.0 is the beginning of the entry string, 1.0 the end.

**xview\_scroll(number, what) :** Scrolls the entry view horizontally by the given amount.

*number* : Number of units.

*what* : What unit to use. This can be either **“units”** (characters) or**“pages”** (larger steps).

# The Variable Classes (BooleanVar, DoubleVar, IntVar, StringVar)

If you program Tk using the Tcl language, you can ask the system to let you know when a variable is changed. The Tk toolkit can use this feature, called*tracing*, to update certain widgets when an associated variable is modified.

There’s no way to track changes to Python variables, but Tkinter allows you to create variable wrappers that can be used wherever Tk can use a traced Tcl variable.

## When to use the Variable Classes

Variables can be used with most entry widgets to track changes to the entered value. The Checkbutton and Radiobutton widgets require variables to work properly.

Variables can also be used to validate the contents of an entry widget, and to change the text in label widgets.

## Patterns

To create a Tkinter variable, call the corresponding constructor:

var = StringVar()

Note that the constructor takes an optional widget argument, but no value argument; to set the value, call the **set** method:

var = StringVar()

var.set("hello")

The constructor argument is only relevant if you’re running Tkinter with multiple Tk instances (which you shouldn’t do, unless you really know what you’re doing).

You can use the **trace** method to attach “observer” callbacks to the variable. The callback is called whenever the contents change:

def callback(\*args):

print ("variable changed!")

var = StringVar()

var.trace("w", callback)

var.set("hello")

## Methods

### get/set

**get()** => value

**set(string)**

The **get** method returns the current value of the variable, as a Python object. For **BooleanVar** variables, the returned value is 0 for false, and 1 for true. For **DoubleVar** variables, the returned value is a Python float. For **IntVar**, it’s an integer. For **StringVar**, it’s either an ASCII string or a Unicode string, depending on the contents.

The **set** method updates the variable, and notifies all variable observers. You can either pass in a value of the right type, or a string.

### trace

**trace(mode, callback)** => string

**trace\_variable(mode, callback)**

Add a variable observer. Returns the internal name of the observer (you can use this to unregister the observer; see below).

The **mode** argument is one of “r” (call observer when variable is read by someone), “w” (call when variable is written by someone), or “u” (undefine; call when the variable is deleted).

### trace\_vdelete

**trace\_vdelete(mode, observer name)**

Remove an observer. The observer name is the string returned by **trace\_variable**, when the observer was first registered.

### trace\_vinfo

**trace\_vinfo()** =>list

# The Tkinter Grid Geometry Manager

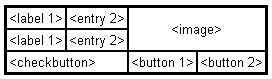
The **Grid** geometry manager puts the widgets in a 2-dimensional table. The master widget is split into a number of rows and columns, and each “cell” in the resulting table can hold a widget.

## When to use the Grid Manager

The grid manager is the most flexible of the geometry managers in Tkinter. If you don’t want to learn how and when to use all three managers, you should at least make sure to learn this one.

The grid manager is especially convenient to use when designing dialog boxes. If you’re using the packer for that purpose today, you’ll be surprised how much easier it is to use the grid manager instead. Instead of using lots of extra frames to get the packing to work, you can in most cases simply pour all the widgets into a single container widget, and use the grid manager to get them all where you want them. (I tend to use two containers; one for the dialog body, and one for the button box at the bottom.)

Consider the following example:



Creating this layout using the pack manager is possible, but it takes a number of extra frame widgets, and a lot of work to make things look good. If you use the grid manager instead, you only need one call per widget to get everything laid out properly (see next section for the code needed to create this layout).

**Warning:** Never mix grid and pack in the same master window. Tkinter will happily spend the rest of your lifetime trying to negotiate a solution that both managers are happy with. Instead of waiting, kill the application, and take another look at your code. A common mistake is to use the wrong parent for some of the widgets.

## Patterns

Using the grid manager is easy. Just create the widgets, and use the **grid** method to tell the manager in which row and column to place them. You don’t have to specify the size of the grid beforehand; the manager automatically determines that from the widgets in it.

Label(master, text="First").grid(row=0)

Label(master, text="Second").grid(row=1)

e1 = Entry(master)

e2 = Entry(master)

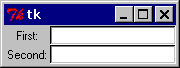
e1.grid(row=0, column=1)

e2.grid(row=1, column=1)

Note that the column number defaults to 0 if not given.

Running the above example produces the following window:

**Simple grid example**



Empty rows and columns are ignored. The result would have been the same if you had placed the widgets in row 10 and 20 instead.

Note that the widgets are centered in their cells. You can use the **sticky** option to change this; this option takes one or more values from the set **N**, **S**, **E**, **W**. To align the labels to the left border, you could use **W** (west):

Label(master, text="First").grid(row=0, sticky=W)

Label(master, text="Second").grid(row=1, sticky=W)

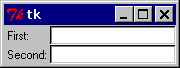
e1 = Entry(master)

e2 = Entry(master)

e1.grid(row=0, column=1)

e2.grid(row=1, column=1)

**Using the sticky option**



You can also have the widgets span more than one cell. The **columnspan** option is used to let a widget span more than one column, and the **rowspan**option lets it span more than one row. The following code creates the layout shown in the previous section:

label1.grid(sticky=E)

label2.grid(sticky=E)

entry1.grid(row=0, column=1)

entry2.grid(row=1, column=1)

checkbutton.grid(columnspan=2, sticky=W)

image.grid(row=0, column=2, columnspan=2, rowspan=2,

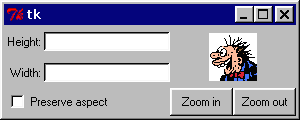
sticky=W+E+N+S, padx=5, pady=5)

button1.grid(row=2, column=2)

button2.grid(row=2, column=3)

There are plenty of things to note in this example. First, no position is specified for the label widgets. In this case, the column defaults to 0, and the row to the *first unused row in the grid*. Next, the entry widgets are positioned as usual, but the checkbutton widget is placed on the next empty row (row 2, in this case), and is configured to span two columns. The resulting cell will be as wide as the label and entry columns combined. The image widget is configured to span both columns and rows at the same time. The buttons, finally, is packed each in a single cell:

**Using column and row spans**



## Reference [#](http://effbot.org/tkinterbook/grid.htm#reference)

**Grid** (class)

Grid geometry manager. This is an implementation class; all the methods described below are available on all widget classes.

**grid(\*\*options)**

Place the widget in a grid as described by the options.

*\*\*options* : Geometry options.

*column=*

Insert the widget at this column. Column numbers start with 0. If omitted, defaults to 0.

*columnspan=*

If given, indicates that the widget cell should span multiple columns. The default is 1.

*in=*

Place widget inside to the given widget. You can only place a widget inside its parent, or in any decendant of its parent. If this option is not given, it defaults to the parent.

Note that **in** is a reserved word in Python. To use it as a keyword option, append an underscore (**in\_**).

*in\_=*

Same as in. See above.

*ipadx=*

Optional horizontal internal padding. Works like **padx**, but the padding is added *inside* the widget borders. Default is 0.

*ipady=*

Optional vertical internal padding. Works like **pady**, but the padding is added *inside* the widget borders. Default is 0.

*padx=*

Optional horizontal padding to place around the widget in a cell. Default is 0.

*pady=*

Optional vertical padding to place around the widget in a cell. Default is 0.

*row=*

Insert the widget at this row. Row numbers start with 0. If omitted, defaults to the first empty row in the grid.

*rowspan=*

If given, indicates that the widget cell should span multiple rows. Default is 1.

*sticky=*

Defines how to expand the widget if the resulting cell is larger than the widget itself. This can be any combination of the constants **S**, **N**, **E**, and **W**, or **NW**, **NE**, **SW**, and **SE**.

For example, **W** (west) means that the widget should be aligned to the left cell border. **W+E** means that the widget should be stretched horizontally to fill the whole cell.**W+E+N+S** means that the widget should be expanded in both directions. Default is to center the widget in the cell.

**grid\_bbox(column=None, row=None, col2=None, row2=None)**

The grid\_bbox method.

*column, row, col2, row2*

**grid\_columnconfigure(index, \*\*options)**

Set options for a cell column.

To change this for a given widget, you have to call this method on the widget’s parent.

*index*

Column index.

*\*\*options*

Column options.

*minsize=*

Defines the minimum size for the column. Note that if a column is completely empty, it will not be displayed, even if this option is set.

*pad=*

Padding to add to the size of the largest widget in the column when setting the size of the whole column.

*weight=*

A relative weight used to distribute additional space between columns. A column with the weight 2 will grow twice as fast as a column with weight 1. The default is 0, which means that the column will not grow at all.

**grid\_configure(\*\*options)**

Same as [**grid**](http://effbot.org/tkinterbook/grid.htm#grid.Grid.grid-method).

**grid\_forget()**

Remove this widget from the grid manager. The widget is not destroyed, and can be displayed again by **grid** or any other manager.

**grid\_info()**

Return a dictionary containing the current cell options for the cell used by this widget.

Returns:

A dictionary containing grid grid management options.

**grid\_location(x, y)**

Returns the grid cell under (or closest to) a given pixel.

*x*

*y*

Returns:

A tuple containing the column and row index.

**grid\_propagate(flag)**

Enables or disables geometry propagation. When enabled, a grid manager connected to this widget attempts to change the size of the widget whenever a child widget changes size. Propagation is always enabled by default.

*flag*

True to enable propagation.

**grid\_remove()**

Remove this widget from the grid manager. The widget is not destroyed, and can be displayed again by **grid** or any other manager.

**grid\_rowconfigure(index, \*\*options)**

Set options for a row of cells.

To change this for a given widget, you have to call this method on the widget’s parent.

*index*

Row index.

*\*\*options*

Row options.

*minsize=*

Defines the minimum size for the row. Note that if a row is completely empty, it will not be displayed, even if this option is set.

*pad=*

Padding to add to the size of the largest widget in the row when setting the size of the whole row.

*weight=*

A relative weight used to distribute additional space between rows. A row with the weight 2 will grow twice as fast as a row with weight 1. The default is 0, which means that the row will not grow at all.

**grid\_size()**

Returns the current grid size for the geometry manager attached to this widget. This is defined as indexes of the first empty column and row in the grid, in that order.

Returns:

A 2-tuple containing the number of columns and rows.

**grid\_slaves(row=None, column=None)**

Returns a list of the “slave” widgets managed by this widget. The widgets are returned as Tkinter widget references.

Returns:

A list of widgets.

**Other Tk modules:**

Other modules that provide Tk support include:

tkinter.scrolledtext: Text widget with a vertical scroll bar built in.

tkinter.colorchooser: Dialog to let the user choose a color.

tkinter.commondialog: Base class for the dialogs defined in the other modules listed here.

tkinter.filedialog: Common dialogs to allow the user to specify a file to open or save.

tkinter.font: Utilities to help work with fonts.

tkinter.messagebox: Access to standard Tk dialog boxes.

tkinter.simpledialog: Basic dialogs and convenience functions.

tkinter.dnd: Drag-and-drop support for tkinter. This is experimental and should become deprecated when it is replaced with the Tk DND.

**Lab Tasks:**

* + - 1. GUI Calculator
      2. Notepad

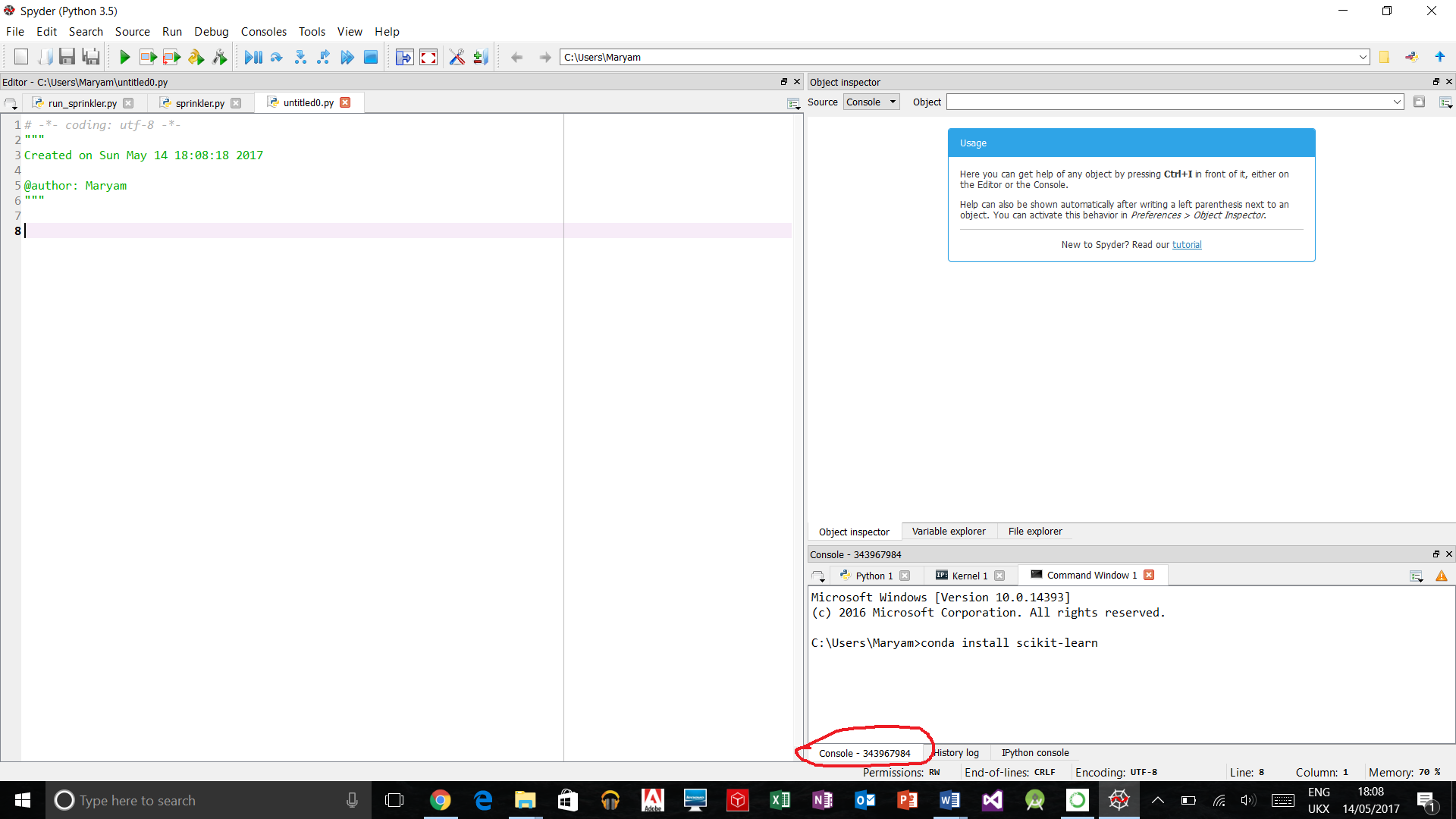
Lab 10

**Machine Learning:**

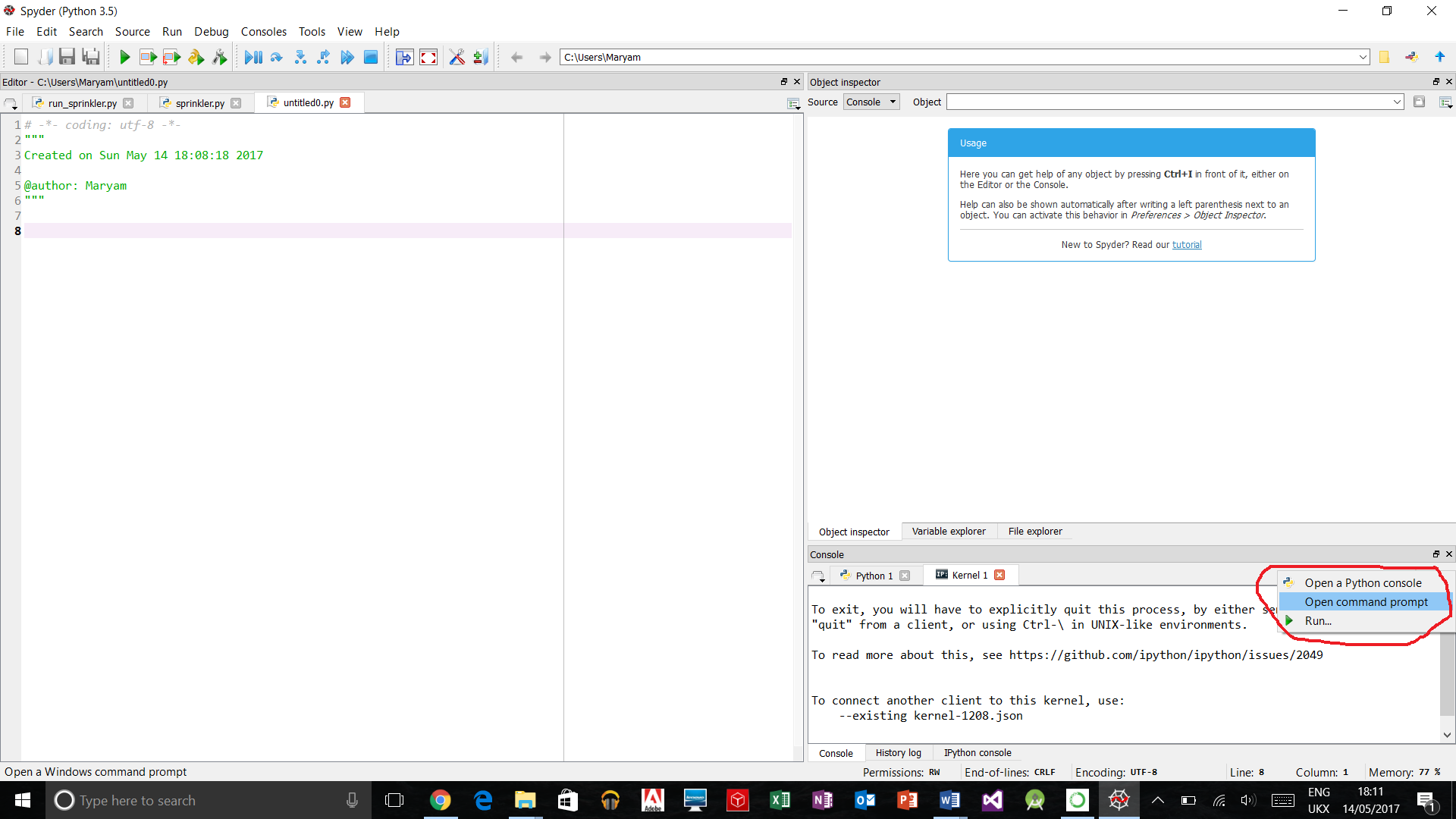
Machine learning is the subfield of computer science that, according to Arthur Samuel in 1959, gives "computers the ability to learn without being explicitly programmed." Evolved from the study of pattern recognition and computational learning theory in artificial intelligence, machine learning explores the study and construction of algorithms that can learn from and make predictions on data – such algorithms overcome following strictly static program instructions by making data-driven predictions or decisions, through building a model from sample inputs. Machine learning is employed in a range of computing tasks where designing and programming explicit algorithms with good performance is difficult or unfeasible; example applications include email filtering, detection of network intruders or malicious insiders working towards a data breach, optical character recognition (OCR), learning to rank and computer vision.

**Install sci-kit package**

Run Anaconda Navigator. Launch Spyder. Select the Console Tab as shown below:

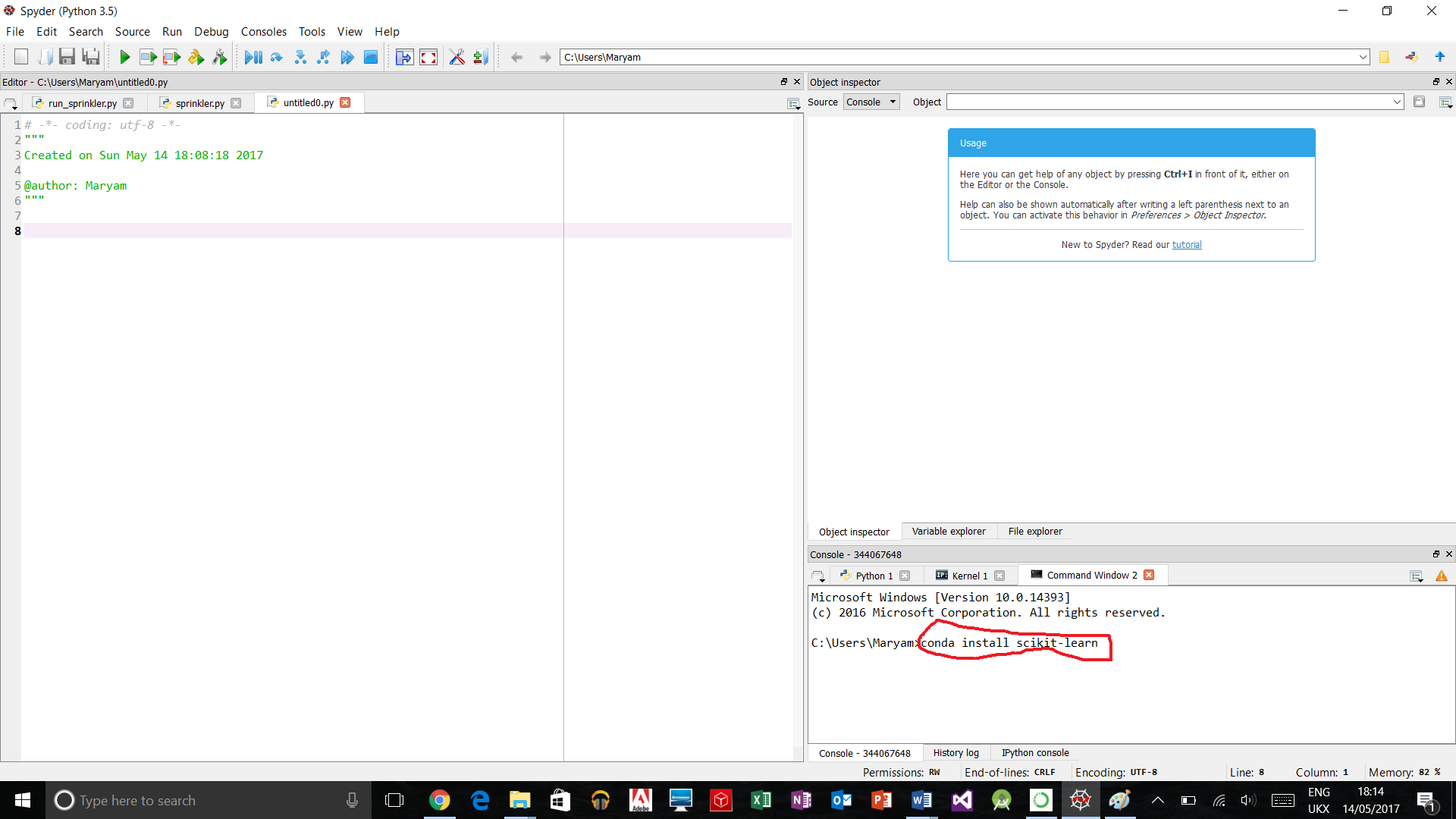


Right click on the title bar of the Console tab and select “Open Command Prompt” as shown :



Now run the command

conda install scikit-learn



When you are prompted whether to proceed or not, type y and press the Enter key.

**Lab Task:**

1. Using KNN determine class of test examples( taken from user) with three different values of K that is one, three and five. Training set is as follows.

|  |  |  |
| --- | --- | --- |
| Feature 1 | Feature 2 | Class |
| 12 | 4 | a |
| 11 | 5 | a |
| 8 | 1 | a |
| 6 | 4 | b |
| 9 | 3 | b |
| 6 | 6 | a |
| 10 | 2 | b |
| 9 | 3 | b |

1. Use data given below to apply k-means clustering. Value for k =2.. Initial mean for cluster 1 and cluster 2 is (5,20, 5) and (10,11,7) respectively.

|  |  |  |
| --- | --- | --- |
| Feature 1 | Feature 2 | Feature 3 |
| 5 | 20 | 4 |
| 6 | 20 | 3 |
| 10 | 10 | 8 |
| 9 | 12 | 9 |
| 8 | 10 | 7 |
| 4 | 22 | 2 |
| 4 | 18 | 4 |
| 12 | 11 | 8 |

Lab 11

**Introduction to Prolog and SWI-Prolog:**

# In this lab, we’ll learn about installing the IDE for a declarative programming language called Prolog.

# Prolog Installation:

# Watch and follow the Prolog Tutorial by Derek Banas:

# <https://youtu.be/SykxWpFwMGs>

**Installing Package Control for Sublime Text 3:**

<https://youtu.be/ioRbV7fQkdU>

Or

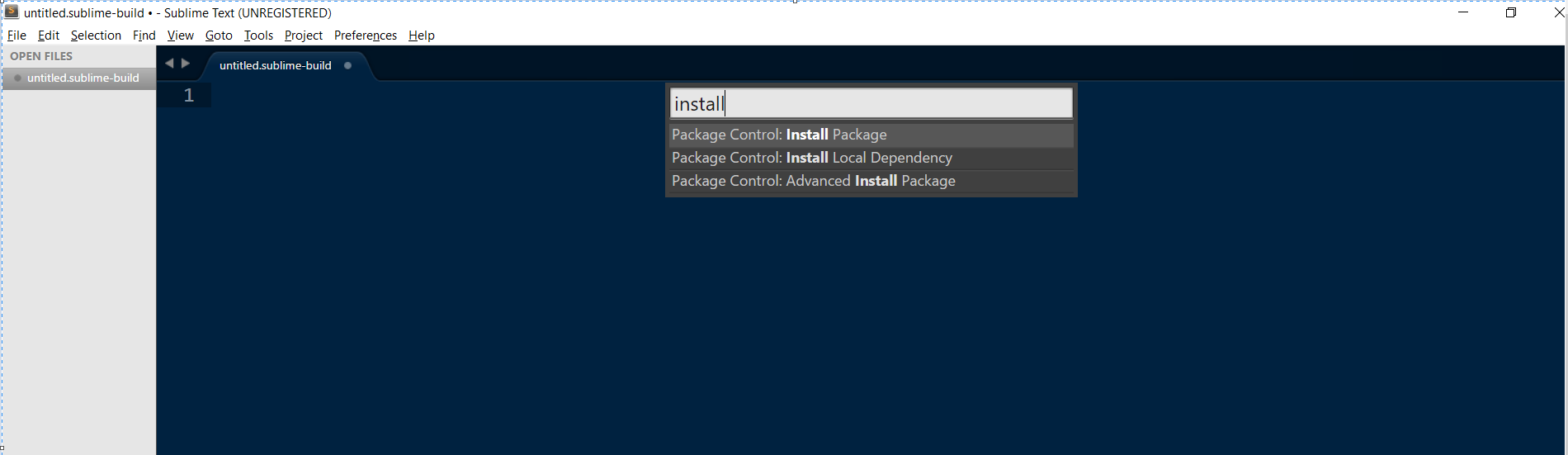
<https://youtu.be/ZN5i-LF3pzU>

**Installing Prolog Package:**

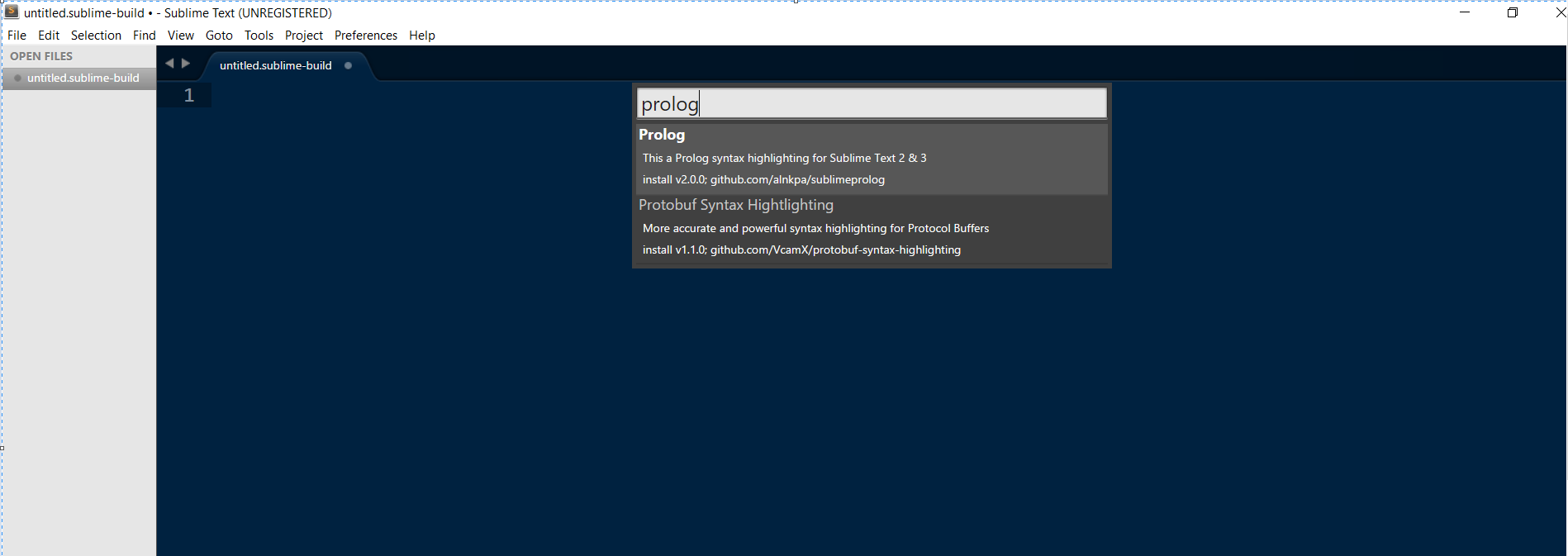
Open Sublime Text 3.

Press Ctrl + Shift + p.

Type Install Package in the window that appears.



Type prolog in the window that appears:



Select prolog syntax highlighting for Sublime Text 2 and 3.

The prolog syntax highlighting package will be installed.

**Installing SWI Prolog and SWI Prolog Editor:**

Installing SWI Prolog:

You can download a stable version of SWI Prolog from:

<http://www.swi-prolog.org/download/stable>

Installing SWI Prolog Editor:

<http://arbeitsplattform.bildung.hessen.de/fach/informatik/swiprolog/indexe.html>

**Lab Task:**

Install SWI-Prolog and SWI-Prolog Editor on your laptops/PCs.

Lab 11 - Part 2 - First Order Logic

# In this lab, we’ll learn about a different way of programming using a declarative programming language called Prolog.

# Introduction to Prolog

There are only three basic constructs in Prolog: facts, rules, and queries. A collection of facts and rules is called a knowledge base (or a database) and Prolog programming is all about writing knowledge bases. That is, **Prolog programs simply** **are** **knowledge bases:** collections of facts and rules which describe some collection of relationships that we find interesting.

So how do we use a Prolog program? By posing queries. That is, by asking questions about the information stored in the knowledge base.

**Syntax table**

|  |  |
| --- | --- |
| **Syntax** | **Meaning** |
| :- | Is implied by  If (implication) |
| , | and (conjunction) |
| ; | or (disjunction) |
| Names starting with capital letters e.g.  X | Variables |
| Names starting with small letters e.g.  woman | Atoms |
| \+ | not |
| < | is less than |
| > | is greater than |
| = | is equal to |
| >= | is greater than or equal to |
| =< | is less than or equal to |
| =:= | equality check |
| =\= | inequality check |

#### Knowledge Base 1

Knowledge Base 1 (KB1) is simply a collection of facts. Facts are used to state things that are unconditionally true of some situation of interest. For example, we can state that Mia, Jody, and Yolanda are women, that Jody plays air guitar, and that a party is taking place, using the following five facts:

  woman(mia).   
 woman(jody).   
 woman(yolanda).   
 playsAirGuitar(jody).   
 party.

This collection of facts is KB1. It is our first example of a Prolog program. Note that the **names** mia, jody, and yolanda, the **properties** woman and playsAirGuitar , and the **proposition** party have been written so that the first letter is in lower-case. **This is important**; we will see why later.

How can we use KB1? By posing queries. That is, by asking questions about the information KB1 contains. Here are some examples. We can ask Prolog whether Mia is a woman by posing the query:

   ?-  woman(mia).

Prolog will answer

   yes

for the obvious reason that this is one of the facts explicitly recorded in KB1. Incidentally, we don’t type in the ?- . This symbol (or something like it, depending on the implementation of Prolog you are using) is the prompt symbol that the Prolog interpreter displays when it is waiting to evaluate a query. We just type in the actual query (for example woman(mia) ) followed by . (a full stop). The full stop is important. If you don’t type it, Prolog won’t start working on the query.

Similarly, we can ask whether Jody plays air guitar by posing the following query:

   ?-  playsAirGuitar(jody).

Prolog will again answer yes, because this is one of the facts in KB1. However, suppose we ask whether Mia plays air guitar:

   ?-  playsAirGuitar(mia).

We will get the answer

   no

Why? Well, first of all, this is not a fact in KB1. Moreover, KB1 is extremely simple, and contains no other information (such as the rules we will learn about shortly) which might help Prolog try to infer (that is, deduce) whether Mia plays air guitar. So Prolog correctly concludes thatplaysAirGuitar(mia) does not follow from KB1.

Here are two important examples. First, suppose we pose the query:

   ?-  playsAirGuitar(vincent).

Again Prolog answers no. Why? Well, this query is about a person (Vincent) that it has no information about, so it (correctly) concludes thatplaysAirGuitar(vincent) cannot be deduced from the information in KB1.

Similarly, suppose we pose the query:

   ?-  tattooed(jody).

Again Prolog will answer no. Why? Well, this query is about a property (being tatooed) that it has no information about, so once again it (correctly) concludes that the query cannot be deduced from the information in KB1. (Actually, some Prolog implementations will respond to this query with an error message, telling you that the predicate or procedure tattooed is not defined; we will soon introduce the notion of predicates.)

Needless to say, we can also make queries concerning propositions. For example, if we pose the query

   ?-  party.

then Prolog will respond

   yes

and if we pose the query

   ?-  rockConcert.

then Prolog will respond

   no

exactly as we would expect.

#### Knowledge Base 2

Here is KB2, our second knowledge base:

   happy(yolanda).   
   listens2Music(mia).   
   listens2Music(yolanda):-  happy(yolanda).   
   playsAirGuitar(mia):-  listens2Music(mia).   
   playsAirGuitar(yolanda):-  listens2Music(yolanda).

There are two facts in KB2, listens2Music(mia) and happy(yolanda) . The last three items it contains are rules.

Rules state information that is conditionally true of the situation of interest. For example, the first rule says that Yolanda listens to music if she is happy, and the last rule says that Yolanda plays air guitar if she listens to music. More generally, the :- should be read as “if”, or “is implied by”. The part on the left hand side of the :- is called the head of the rule, the part on the right hand side is called the body. So in general rules say: if the body of the rule is true, then the head of the rule is true too. And now for the key point:

If a knowledge base contains a rule head  :-  body, and Prolog knows that body follows from the information in the knowledge base, then Prolog can infer head.

This fundamental deduction step is called modus ponens.

Let’s consider an example. Suppose we ask whether Mia plays air guitar:

   ?-  playsAirGuitar(mia).

Prolog will respond yes. Why? Well, although it can’t find playsAirGuitar(mia) as a fact explicitly recorded in KB2, it can find the rule

   playsAirGuitar(mia):-  listens2Music(mia).

Moreover, KB2 also contains the fact listens2Music(mia) . Hence Prolog can use the rule of modus ponens to deduce thatplaysAirGuitar(mia) .

Our next example shows that Prolog can chain together uses of modus ponens. Suppose we ask:

   ?-  playsAirGuitar(yolanda).

Prolog would respond yes. Why? Well, first of all, by using the fact happy(yolanda) and the rule

   listens2Music(yolanda):-  happy(yolanda).

Prolog can deduce the new fact listens2Music(yolanda) . This new fact is not explicitly recorded in the knowledge base — it is only implicitlypresent (it is inferred knowledge). Nonetheless, Prolog can then use it just like an explicitly recorded fact. In particular, from this inferred fact and the rule

   playsAirGuitar(yolanda):-  listens2Music(yolanda).

it can deduce playsAirGuitar(yolanda) , which is what we asked it. Summing up: any fact produced by an application of modus ponens can be used as input to further rules. By chaining together applications of modus ponens in this way, Prolog is able to retrieve information that logically follows from the rules and facts recorded in the knowledge base.

The facts and rules contained in a knowledge base are called clauses. Thus KB2 contains five clauses, namely three rules and two facts. Another way of looking at KB2 is to say that it consists of three predicates (or procedures). The three **predicates** are:

   listens2Music   
   happy   
   playsAirGuitar

The happy predicate is defined using a single clause (a fact). The listens2Music and playsAirGuitar predicates are each defined using two clauses (in one case, two rules, and in the other case, one rule and one fact). It is a good idea to think about Prolog programs in terms of the predicates they contain. In essence, the predicates are the concepts we find important, and the various clauses we write down concerning them are our attempts to pin down what they mean and how they are inter-related.

One final remark. **We can view a fact as a rule with an empty body**. That is, we can think of facts as conditionals that do not have any antecedent conditions, or degenerate rules.

#### Knowledge Base 3

KB3, our third knowledge base, consists of five clauses:

   happy(vincent).   
   listens2Music(butch).   
   playsAirGuitar(vincent):-   
         listens2Music(vincent),   
         happy(vincent).   
   playsAirGuitar(butch):-   
         happy(butch).   
   playsAirGuitar(butch):-   
         listens2Music(butch).

There are two facts, happy(vincent) and listens2Music(butch) , and three rules.

KB3 defines the same three predicates as KB2 (namely happy , listens2Music , and playsAirGuitar ) but it defines them differently. In particular, the three rules that define the playsAirGuitar predicate introduce some new ideas. First, note that the rule

   playsAirGuitar(vincent):-   
         listens2Music(vincent),   
         happy(vincent).

has two items in its body, or (to use the standard terminology) two goals. So, what exactly does this rule mean? The most important thing to note is the comma , that separates the goal listens2Music(vincent) and the goal happy(vincent) in the rule’s body. This is the way logical conjunction is expressed in Prolog (that is, the comma means and ). So this rule says: “Vincent plays air guitar if he listens to music and he is happy”.

Thus, if we posed the query

   ?-  playsAirGuitar(vincent).

Prolog would answer no. This is because while KB3 contains happy(vincent) , it does not explicitly contain the informationlistens2Music(vincent) , and this fact cannot be deduced either. So KB3 only fulfils one of the two preconditions needed to establishplaysAirGuitar(vincent) , and our query fails.

Incidentally, the spacing used in this rule is irrelevant. For example, we could have written it as

   playsAirGuitar(vincent):-  happy(vincent),   
                          listens2Music(vincent).

and it would have meant exactly the same thing. Prolog offers us a lot of freedom in the way we set out knowledge bases, and we can take advantage of this to keep our code readable.

Next, note that KB3 contains two rules with exactly the same head, namely:

   playsAirGuitar(butch):-   
         happy(butch).   
   playsAirGuitar(butch):-   
         listens2Music(butch).

This is a way of stating that Butch plays air guitar either if he listens to music, or if he is happy. That is, listing multiple rules with the same head is a way of expressing logical disjunction (that is, it is a way of saying or ). So if we posed the query

   ?-  playsAirGuitar(butch).

Prolog would answer yes. For although the first of these rules will not help (KB3 does not allow Prolog to conclude that happy(butch) ), KB3does contain listens2Music(butch) and this means Prolog can apply modus ponens using the rule

   playsAirGuitar(butch):-   
         listens2Music(butch).

to conclude that playsAirGuitar(butch) .

There is another way of **expressing disjunction** in Prolog. We could replace the pair of rules given above by the single rule

   playsAirGuitar(butch):-   
         happy(butch);   
         listens2Music(butch).

That is, the **semicolon** **;** is the Prolog symbol for **or** , so this single rule means exactly the same thing as the previous pair of rules. Is it better to use multiple rules or the semicolon? That depends. On the one hand, extensive use of semicolon can make Prolog code hard to read. On the other hand, the semicolon is more efficient as Prolog only has to deal with one rule.

It should now be clear that **Prolog has something to do with logic**: after all, the **:-** **means implication**, the **,** **means conjunction**, and the **;** **means disjunction**. (What about **negation**? That is a whole other story.) Moreover, we have seen that a standard logical proof rule (**modus ponens**) plays an important role in Prolog programming. So we are already beginning to understand why “Prolog” is short for “Programming with logic”.

#### Knowledge Base 4

Here is KB4, our fourth knowledge base:

   woman(mia).   
   woman(jody).   
   woman(yolanda).   
      
   loves(vincent,mia).   
   loves(marsellus,mia).   
   loves(pumpkin,honey\_bunny).   
   loves(honey\_bunny,pumpkin).

Now, this is a pretty boring knowledge base. There are no rules, only a collection of facts. Ok, we are seeing a relation that has two names as arguments for the first time (namely the loves relation), but, let’s face it, that’s a rather predictable idea.

No, the novelty this time lies not in the knowledge base, it lies in the queries we are going to pose. In particular, for the first time we’re going to make use of variables . Here’s an example:

   ?-  woman(X).

The X is a variable (in fact, any word beginning with an upper-case letter is a Prolog variable, which is why we had to be careful to use lower-case initial letters in our earlier examples). Now **a variable isn’t a name, rather it’s a** **placeholder** **for information**. That is, ***this query asks Prolog: tell me which of the individuals you know about is a woman.***

Prolog answers this query by working its way through KB4, from top to bottom, trying to unify (or match) the expression woman(X) with the information KB4 contains. Now the first item in the knowledge base is woman(mia) . So, Prolog unifies X with mia , thus making the query agree perfectly with this first item. (Incidentally, there’s a lot of different terminology for this process: we can also say that Prolog instantiates X to mia , or that it binds X to mia .) Prolog then reports back to us as follows:

   X  =  mia

That is, it not only says that there is information about at least one woman in KB4, it actually tells us who she is. It didn’t just say yes, it actually gave us the variable binding (or variable instantiation) that led to success.

But that’s not the end of the story. The whole point of variables is that they can stand for, or unify with, different things. And there is information about other women in the knowledge base. We can access this information by typing a semicolon:

   X  =  mia  ;

Remember that ; means or , so this query means: are there any alternatives ? So Prolog begins working through the knowledge base again (it remembers where it got up to last time and starts from there) and sees that if it unifies X with jody , then the query agrees perfectly with the second entry in the knowledge base. So it responds:

   X  =  mia  ;   
   X  =  jody

It’s telling us that there is information about a second woman in KB4, and (once again) it actually gives us the value that led to success. And of course, if we press ; a second time, Prolog returns the answer

   X  =  mia  ;   
   X  =  jody  ;   
   X  =  yolanda

But what happens if we press ; a third time? Prolog responds no. No other unifications are possible. There are no other facts starting with the symbol woman . The last four entries in the knowledge base concern the love relation, and there is no way that such entries can be unified with a query of the form woman(X) .

Let’s try a more complicated query, namely

   ?-  loves(marsellus,X),  woman(X).

Now, remember that , means and , so this query says: is there any individual X such that Marsellus loves X and X is a woman ? If you look at the knowledge base you’ll see that there is: Mia is a woman (fact 1) and Marsellus loves Mia (fact 5). And in fact, Prolog is capable of working this out. That is, it can search through the knowledge base and work out that if it unifies X with Mia, then both conjuncts of the query are satisfied. So Prolog returns the answer

   X  =  mia

The business of unifying variables with information in the knowledge base is the heart of Prolog. As we’ll learn, there are many interesting ideas in Prolog — but when you get right down to it, it’s Prolog’s ability to perform unification and return the values of the variable bindings to us that is crucial.

#### Knowledge Base 5

Well, we’ve introduced variables, but so far we’ve only used them in queries. But variables not only can be used in knowledge bases, it’s only when we start to do so that we can write truly interesting programs. Here’s a simple example, the knowledge base KB5:

   loves(vincent,mia).   
   loves(marsellus,mia).   
   loves(pumpkin,honey\_bunny).   
   loves(honey\_bunny,pumpkin).   
      
   jealous(X,Y):-  loves(X,Z),  loves(Y,Z).

KB5 contains four facts about the loves relation and one rule. (Incidentally, the blank line between the facts and the rule has no meaning: it’s simply there to increase the readability. As we said earlier, Prolog gives us a great deal of freedom in the way we format knowledge bases.) But this rule is by far the most interesting one we have seen so far: it contains three variables (note that X , Y , and Z are all upper-case letters). What does it say?

In effect, it is defining a concept of jealousy. It says that an individual X will be jealous of an individual Y if there is some individual Z that X loves, and Y loves that same individual Z too. (Ok, so jealousy isn’t as straightforward as this in the real world.) The key thing to note is that this is a general statement: it is not stated in terms of mia , or pumpkin , or anyone in particular — it’s a conditional statement about everybody in our little world.

Suppose we pose the query:

   ?-  jealous(marsellus,W).

This query asks: can you find an individual W such that Marsellus is jealous of W ? Vincent is such an individual. If you check the definition of jealousy, you’ll see that Marsellus must be jealous of Vincent, because they both love the same woman, namely Mia. So Prolog will return the value

   W  =  vincent

## Prolog Syntax

Now that we’ve got some idea of what Prolog does, it’s time to go back to the beginning and work through the details more carefully. Let’s start by asking a very basic question: we’ve seen all kinds of expressions (for example jody , playsAirGuitar(mia) , and X ) in our Prolog programs, but these have just been examples. It’s time for precision: exactly what are facts, rules, and queries built out of?

The answer is terms, and there are four kinds of term in Prolog: atoms, numbers, variables, and complex terms (or structures). Atoms and numbers are lumped together under the heading constants, and constants and variables together make up the simple terms of Prolog.

Let’s take a closer look. To make things crystal clear, let’s first be precise about the basic characters (that is, symbols) at our disposal. The upper-case letters are A , B ,…, Z ; the lower-case letters are a , b ,…, z ; the digits are 0 , 1 , 2 ,…, 9 . In addition we have the \_ symbol, which is called underscore, and some special characters , which include characters such as + , - , \* , / , < , > , = , : , . , & , ~ . The blank space is also a character, but a rather unusual one, being invisible. A string is an unbroken sequence of characters.

#### Atoms

An atom is either:

1. A string of characters made up of upper-case letters, lower-case letters, digits, and the underscore character, that begins with a lower-case letter. Here are some examples: butch , big\_kahuna\_burger , listens2Music and playsAirGuitar .
2. An arbitrary sequence of characters enclosed in single quotes. For example ‘Vincent ’, ‘The  Gimp ’, ‘Five\_Dollar\_Shake ’, ‘&^%&#@$  &\* ’, and ‘   ’. The sequence of characters between the single quotes is called the atom name. Note that **we are allowed to use spaces in such atoms; in fact, a common reason for using single quotes is so we can do precisely that.**
3. A string of special characters. Here are some examples: @= and ====> and ; and :- are all atoms. As we have seen, some of these atoms, such as ; and :- have a pre-defined meaning.

#### Numbers

Real numbers aren’t particularly important in typical Prolog applications. So although most Prolog implementations do support floating point numbers or floats (that is, representations of real numbers such as 1657.3087 or π ) we say little about them in this book.

But integers (that is: …,-2, -1, 0, 1, 2, 3,…) are useful for such tasks as counting the elements of a list, and we’ll discuss how to manipulate them in Chapter  [5](http://www.learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlch5). Their Prolog syntax is the obvious one: 23, 1001, 0, -365, and so on.

#### Variables

**A variable is a string of upper-case letters, lower-case letters, digits and underscore characters that starts** **either** **with an upper-case letter** **or with an underscore**. For example, X , Y , Variable , \_tag , X\_526 , List , List24 , \_head , Tail , \_input and Output are all Prolog variables.

The variable \_ (that is, a single underscore character) is rather special. It’s called the anonymous variable , and we discuss it in Chapter  [4](http://www.learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlch4).

#### Complex terms

Constants, numbers, and variables are the building blocks: now we need to know how to fit them together to make complex terms. Recall that complex terms are often called structures.

**Complex terms are build out of a functor followed by a sequence of arguments**. The arguments are put in ordinary parentheses, separated by commas, and placed after the functor. Note that the functor has to be directly followed by the parenthesis; you can’t have a space between the functor and the parenthesis enclosing the arguments. The functor must be an atom. That is, variables cannot be used as functors. On the other hand, arguments can be any kind of term.

Now, we’ve already seen lots of examples of complex terms when we looked at the knowledge bases KB1 to KB5. For example, playsAirGuitar(jody) is a complex term: its functor is playsAirGuitar and its argument is jody . Other examples are loves(vincent,mia) and, to give an example containing a variable, jealous(marsellus,W) .

But the definition allows for more complex terms than this. In fact, it allows us to keep nesting complex terms inside complex terms indefinitely (that is, it is allows recursive structure). For example

   hide(X,father(father(father(butch))))

is a perfectly acceptable complex term. Its functor is hide , and it has two arguments: the variable X , and the complex term father(father(father(butch))) . This complex term has father as its functor, and another complex term, namely father(father(butch)) , as its sole argument. And the argument of this complex term, namely father(butch) , is also complex. But then the nesting bottoms out, for the argument here is the constant butch .

As we shall see, such nested (or recursively structured) terms enable us to represent many problems naturally. In fact, **the interplay between recursive term structure and variable unification is the source of much of Prolog’s power**.

**Arity:**

The **number of arguments that a complex term has is called its arity**. For example, woman(mia) is a complex term of arity 1, and loves(vincent, mia) is a complex term of arity 2.

Arity is important to Prolog. Prolog would be quite happy for us to define two predicates with the same functor but with a different number of arguments. For example, we are free to define a knowledge base that defines a two-place predicate love (this might contain such facts as love(vincent,mia) ), and also a three-place love predicate (which might contain such facts as love(vincent,marsellus,mia) ). However, if we did this, Prolog would treat the two-place love and the three-place love as different predicates. Later we shall see that it can be useful to define two predicates with the same functor but different arity.

When we need to talk about predicates and how we intend to use them (for example, in documentation) it is usual to use a suffix / followed by a number to indicate the predicate’s arity. To return to KB2, instead of saying that it defines predicates

   listens2Music   
   happy   
   playsAirGuitar

we should really say that it defines predicates

   listens2Music/1   
   happy/1   
   playsAirGuitar/1

And Prolog can’t get confused about a knowledge base containing the two different love predicates, for it regards the love/2 predicate and the love/3 predicate as distinct.

**Lab Task:**

1. Are there any other jealous people in KB5?
2. Suppose we wanted Prolog to tell us about all the jealous people: what query would we pose? Do any of the answers surprise you? Do any seem silly?
3. Which of the following sequences of characters are atoms, which are variables, and which are neither?
   1. vINCENT
   2. Footmassage
   3. variable23
   4. Variable2000
   5. big\_kahuna\_burger
   6. ’big  kahuna  burger’
   7. big  kahuna  burger
   8. ’Jules’
   9. \_Jules
   10. ’\_Jules’
4. Which of the following sequences of characters are atoms, which are variables, which are complex terms, and which are not terms at all? Give the functor and arity of each complex term.
   1. loves(Vincent,mia)
   2. ’loves(Vincent,mia)’
   3. Butch(boxer)
   4. boxer(Butch)
   5. and(big(burger), kahuna(burger))
   6. and(big(X), kahuna(X))
   7. \_and(big(X), kahuna(X))
   8. (Butch  kills  Vincent)
   9. kills(Butch  Vincent)
   10. kills(Butch,Vincent)
5. How many facts, rules, clauses, and predicates are there in the following knowledge base? What are the heads of the rules, and what are the goals they contain?

woman(vincent).

woman(mia).

man(jules).

person(X):-  man(X);  woman(X).

loves(X,Y):-  father(X,Y).

father(Y,Z):-  man(Y),  son(Z,Y).

father(Y,Z):-  man(Y),  daughter(Z,Y).

1. Represent the following in Prolog:
   1. Butch is a killer.
   2. Mia and Marsellus are married.
   3. Zed is dead.
   4. Marsellus kills everyone who gives Mia a foot massage.
   5. Mia loves everyone who is a good dancer.
   6. Jules eats anything that is nutritious or tasty.
2. Suppose we are working with the following knowledge base:

wizard(ron).   
hasWand(harry).   
quidditchPlayer(harry).   
wizard(X):-  hasBroom(X),  hasWand(X).   
hasBroom(X):-  quidditchPlayer(X).

How does Prolog respond to the following queries?

* 1. wizard(ron).
  2. witch(ron).
  3. wizard(hermione).
  4. witch(hermione).
  5. wizard(harry).
  6. wizard(Y).
  7. witch(Y).

Lab 12

**Unification, Recursion, First Order Logic, Expert Systems:**

**Unification:**

In Prolog, two terms unify:

* if they are the same term, or
* if they contain variables that can be uniformly instantiated with terms in such a way that the resulting terms are equal

When trying to unify terms, Prolog scans the knowledge base top down.

**Recursive Definitions**

Predicates can be defined recursively. Roughly speaking, a predicate is recursively defined if one or more rules in its definition refers to itself.

Let’s consider an example. Suppose we have a knowledge base recording facts about the child relation:

   child(bridget,caroline).   
   child(caroline,donna).

That is, Caroline is a child of Bridget, and Donna is a child of Caroline. Now suppose we wished to define the descendant relation; that is, the relation of being a child of, or a child of a child of, or a child of a child of a child of, and so on. Here’s a first attempt to do this. We could add the following two non -recursive rules to the knowledge base:

   descend(X,Y)  :-  child(X,Y).   
      
   descend(X,Y)  :-  child(X,Z),   
                     child(Z,Y).

Now, fairly obviously these definitions work up to a point, but they are clearly limited: they only define the concept of descendant-of for two generations or less. That’s ok for the above knowledge base, but suppose we get some more information about the child-of relation and we expand our list of child-of facts to this:

   child(anne,bridget).   
   child(bridget,caroline).   
   child(caroline,donna).   
   child(donna,emily).

Now our two rules are inadequate. For example, if we pose the queries

   ?-  descend(anne,donna).

or

   ?-  descend(bridget,emily).

we get the answer no, which is not what we want. Sure, we could ‘fix’ this by adding the following two rules:

   descend(X,Y)  :-  child(X,Z\_1),   
                                     child(Z\_1,Z\_2),   
                                     child(Z\_2,Y).   
      
   descend(X,Y)  :-  child(X,Z\_1),   
                                     child(Z\_1,Z\_2),   
                                     child(Z\_2,Z\_3),   
                                     child(Z\_3,Y).

But, let’s face it, this is clumsy and hard to read. Moreover, if we add further child-of facts, we could easily find ourselves having to add more and more rules as our list of child-of facts grow, rules like:

   descend(X,Y)  :-  child(X,Z\_1),   
                                     child(Z\_1,Z\_2),   
                                     child(Z\_2,Z\_3),   
                                                                                 .   
                                             .   
                                             .   
                                     child(Z\_17,Z\_18).   
                                     child(Z\_18,Z\_19).   
                                     child(Z\_19,Y).

This is not a particularly pleasant (or sensible) way to go!

But we don’t need to do this at all. We can avoid having to use ever longer rules entirely. The following recursive predicate definition fixes everything exactly the way we want:

   descend(X,Y)  :-  child(X,Y).   
      
   descend(X,Y)  :-  child(X,Z),   
                     descend(Z,Y).

What does this say? The declarative meaning of the base clause is: if Y is a child of X , then Y is a descendant of X . Obviously sensible. So what about the recursive clause? Its declarative meaning is: if Z is a child of X , and Y is a descendant of Z , then Y is a descendant of X . Again, this is obviously true.

So let’s now look at the procedural meaning of this recursive predicate, by stepping through an example. What happens when we pose the query:

   descend(anne,donna)

Prolog first tries the first rule. The variable X in the head of the rule is unified with anne and Y with donna and the next goal Prolog tries to prove is

   child(anne,donna)

This attempt fails, however, since the knowledge base neither contains the fact child(anne,donna) nor any rules that would allow to infer it. So Prolog backtracks and looks for an alternative way of proving descend(anne,donna) . It finds the second rule in the knowledge base and now has the following subgoals:

   child(anne,\_633),   
   descend(\_633,donna).

Prolog takes the first subgoal and tries to unify it with something in the knowledge base. It finds the fact child(anne,bridget) and the variable \_633 gets instantiated to bridget . Now that the first subgoal is satisfied, Prolog moves to the second subgoal. It has to prove

   descend(bridget,donna)

This is the first recursive call of the predicate descend/2 . As before, Prolog starts with the first rule, but fails, because the goal

   child(bridget,donna)

cannot be proved. Backtracking, Prolog finds that there is a second possibility to be checked for descend(bridget,donna) , namely the second rule, which again gives Prolog two new subgoals:

   child(bridget,\_1785),   
   descend(\_1785,donna).

The first one can be unified with the fact child(bridget,caroline) of the knowledge base, so that the variable \_1785 is instantiated with caroline . Next Prolog tries to prove

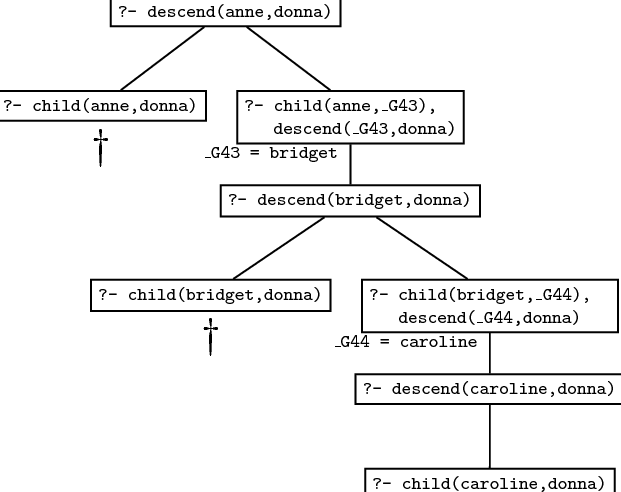
   descend(caroline,donna).

This is the second recursive call of predicate descend/2 . As before, it tries the first rule first, obtaining the following new goal:

   child(caroline,donna)

This time Prolog succeeds, since child(caroline,donna) is a fact in the database. Prolog has found a proof for the goal descend(caroline,donna) (the second recursive call). But this means that descend(bridget,donna) (the first recursive call) is also true, which means that our original query descend(anne,donna) is true as well.

Here is the search tree for the query descend(anne,donna) . Make sure that you understand how it relates to the discussion in the text; that is, how Prolog traverses this search tree when trying to prove this query.



It should be obvious from this example that no matter how many generations of children we add, we will always be able to work out the descendant relation. That is, the recursive definition is both general and compact: it contains all the information in the non-recursive rules, and much more besides. The non-recursive rules only defined the descendant concept up to some fixed number of generations: we would need to write down infinitely many non-recursive rules if we wanted to capture this concept fully, and of course that’s impossible. But, in effect, that’s what the recursive rule does for us: it bundles up the information needed to cope with arbitrary numbers of generations into just three lines of code.

Recursive rules are really important. They enable to pack an enormous amount of information into a compact form and to define predicates in a natural way. Most of the work you will do as a Prolog programmer will involve writing recursive rules.

**Lab Task:**

1. medical diagnostic system
2. university admission and scholarship knowledge base

Lab 13

**File Handling:**

**Syntax:**

[open()](https://docs.python.org/3.4/library/functions.html#open) returns a [file object](https://docs.python.org/3.4/glossary.html#term-file-object), and is most commonly used with two arguments: open(filename, mode). For example:

f = open('pathtofile', 'r')

f.read()

mode can have the following values:

* 'r' for read only,
* 'w' for writing only (if there is an old file, it will be written over),
* 'a' for appending (adding things on to the end of the file)
* 'r+' for both reading and writing.

Normally, files are opened in *text mode*, that means, you read and write strings from and to the file, which are encoded in a specific encoding. If encoding is not specified, the default is platform dependent (see [open()](https://docs.python.org/3.4/library/functions.html#open)). 'b' appended to the mode opens the file in *binary mode*: now the data is read and written in the form of bytes objects. This mode should be used for all files that don’t contain text.

In text mode, the default when reading is to convert platform-specific line endings (\n on Unix, \r\n on Windows) to just \n. When writing in text mode, the default is to convert occurrences of \n back to platform-specific line endings. This behind-the-scenes modification to file data is fine for text files, but will corrupt binary data like that in JPEG or EXE files. Be very careful to use binary mode when reading and writing such files.

**Methods of file Objects:**

**f.read()**

To read a file’s contents, call f.read(size), which reads some quantity of data and returns it as a string or bytes object. *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned; it’s your problem if the file is twice as large as your machine’s memory. Otherwise, at most *size* bytes are read and returned. If the end of the file has been reached, f.read() will return an empty string ('').

**f.readline()**

f.readline() reads a single line from the file; a newline character (\n) is left at the end of the string, and is only omitted on the last line of the file if the file doesn’t end in a newline. This makes the return value unambiguous; if f.readline() returns an empty string, the end of the file has been reached, while a blank line is represented by '\n', a string containing only a single newline.

**f.readlines()**

The **readlines()** method returns a list of remaining lines of the entire file.

All these reading methods return empty values when end of file (EOF) is reached.

**Loops, list, and f.readlines()**

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

**>>> for** line **in** f:

**...**  print(line, end='')

**...**

This is the first line of the file.

Second line of the file

If you want to read all the lines of a file in a list you can also use list(f) or f.readlines().

**f.write(string)**

f.write(string) writes the contents of *string* to the file, returning the number of characters written. To write something other than a string, it needs to be converted to a string first:

**>>>** value = ('the answer', 42)

**>>>** s = str(value)

**>>>** f.write(s)

18

f.tell() returns an integer giving the file object’s current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.

To change the file object’s position, use f.seek(offset, from\_what). The position is computed from adding offset to a reference point; the reference point is selected by the from\_what argument. A from\_what value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. from\_what can be omitted and defaults to 0, using the beginning of the file as the reference point.

**>>>** f = open('workfile', 'rb+')

**>>>** f.write(b'0123456789abcdef')

16

**>>>** f.seek(5) *# Go to the 6th byte in the file*

5

**>>>** f.read(1)

b'5'

**>>>** f.seek(-3, 2) *# Go to the 3rd byte before the end*

13

**>>>** f.read(1)

b'd'

**f.seek() and f.tell()**

In text files (those opened without a b in the mode string), only seeks relative to the beginning of the file are allowed (the exception being seeking to the very file end with seek(0, 2)) and the only valid offset values are those returned from the f.tell(), or zero. Any other offset value produces undefined behaviour.

**f.close()**

When you’re done with a file, call f.close() to close it and free up any system resources taken up by the open file. After calling f.close(), attempts to use the file object will automatically fail.

**>>>** f.close()

**>>>** f.read()

Traceback (most recent call last):

File "<stdin>", line 1, in ?

ValueError: I/O operation on closed file

**with keyword:**

It is good practice to use the with keyword when dealing with file objects. This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way. It is also much shorter than writing equivalent try-finally blocks:

**>>> with** open('workfile', 'r') **as** f:

**...**  read\_data = f.read()

**>>>** f.closed

True

**seek(offset, whence)**

whence is optional, and determines where to seek from. If whence is 0, the bytes/letters are counted from the beginning. If it is 1, the bytes are counted from the current cursor position. If it is 2, then the bytes are counted from the end of the file. If nothing is put there, 0 is assumed.

offset describes how far from whence that the cursor moves. for example:

* openfile.seek(45,0) would move the cursor to 45 bytes/letters after the beginning of the file.
* openfile.seek(10,1) would move the cursor to 10 bytes/letters after the current cursor position.
* openfile.seek(-77,2) would move the cursor to 77 bytes/letters before the end of the file (notice the - before the 77)

**Regular Expression:**

Pattern that a string is searched for, in python regular expression is first compiled.

Keyword = re.compile(r “the”)

Then the search for the regular expression in a line is executed this way:

Keyword.search(line)

More information can be found on <https://docs.python.org/3.4/library/re.html>

**File Handling in Tkinter:**

If you want to open or save a file or to choose a directory using a filedialog you don’t need to implement it on your own. The module filedialog is just for this purpose. In most cases the seven convenience functions provided by the module will serve your needs.

**Functions**

First you have to decide if you want to open a file or just want to get a filename in order to open the file on your own. In the first case you should use fileDialog.askopenfile() in the latter case fileDialog.askopenfilename(). Both functions come in a multiple file version with just the same parameters as the single file version which allow the user to select multiple files.

The multiple file versions return a list of open files or a list of file names. If no files were selected or the cancel button was pressed an empty list is returned.

Saving files works in a similar way. You also have two variants of the function, one to get an opened file which you can use to save your data and another to get a file name in order to open the file on your own. These functions are only provided in the single file version. A multiple file version would make no sense.

|  |  |  |  |
| --- | --- | --- | --- |
|  | single file | multiple files | available options |
| open a file | askopenfile(mode='r', \*\*options) | askopenfiles(mode='r', \*\*options) | defaultextension, filetypes, initialdir, initialfile, multiple, message, parent, title |
| get a filename to open | askopenfilename(\*\*options) | askopenfilenames(\*\*options) | defaultextension, filetypes, initialdir, initialfile, multiple, message, parent, title |
| save a file | asksaveasfile(mode='w', \*\*options) | n/a | defaultextension, filetypes, initialdir, initialfile, multiple, message, parent, title |
| get a filename to save | asksaveasfilename(\*\*options) | n/a | defaultextension, filetypes, initialdir, initialfile, multiple, message, parent, title |
| choose a directory | askdirectory(\*\*options) | n/a | initialdir, parent, title, mustexist |

## Options

-defaultextension extension

* Specifies a string that will be appended to the filename if the user enters a filename without an extension. The default value is the empty string, which means no extension will be appended to the filename in any case. This option is ignored on the Macintosh platform, which does not require extensions to filenames, and the UNIX implementation guesses reasonable values for this from the -filetypes option when this is not supplied.

-filetypes filePatternList

* If a File types listbox exists in the file dialog on the particular platform, this option gives the filetypes in this listbox. When the user choose a filetype in the listbox, only the files of that type are listed. If this option is unspecified, or if it is set to the empty list, or if the File types listbox is not supported by the particular platform then all files are listed regardless of their types. See the section SPECIFYING FILE PATTERNS below for a discussion on the contents of filePatternList.

-initialdir directory

* Specifies that the files in directory should be displayed when the dialog pops up. If this parameter is not specified, then the files in the current working directory are displayed. If the parameter specifies a relative path, the return value will convert the relative path to an absolute path. This option may not always work on the Macintosh. This is not a bug. Rather, the General Controls control panel on the Mac allows the end user to override the application default directory.

-initialfile filename

* Specifies a filename to be displayed in the dialog when it pops up. This option is ignored on the Macintosh platform.

-message string

* Specifies a message to include in the client area of the dialog. This is only available on the Macintosh, and only when Navigation Services are installed.

-multiple boolean

* Allows the user to choose multiple files from the Open dialog. On the Macintosh, this is only available when Navigation Services are installed.

-mustexist boolean

* Specifies whether the user may specify non-existent directories. If this parameter is true, then the user may only select directories that already exist. The default value is false.

-parent window

* Makes window the logical parent of the dialog. The dialog is displayed on top of its parent window.

-title titleString

* Specifies a string to display as the title of the dialog box. If this option is not specified, then a default title will be displayed.

## Example

This example just should give you an idea of the modules use.

import tkinter, tkinter.constants, tkinter.filedialog

class fileDialogExample(tkinter.Frame):

def \_\_init\_\_(self, root):

tkinter.Frame.\_\_init\_\_(self, root)

# options for buttons

button\_opt = {'fill': tkinter.constants.BOTH, 'padx': 5, 'pady': 5}

# define buttons

tkinter.Button(self, text='askopenfile', command=self.askopenfile).pack(\*\*button\_opt)

tkinter.Button(self, text='askopenfilename', command=self.askopenfilename).pack(\*\*button\_opt)

tkinter.Button(self, text='asksaveasfile', command=self.asksaveasfile).pack(\*\*button\_opt)

tkinter.Button(self, text='asksaveasfilename', command=self.asksaveasfilename).pack(\*\*button\_opt)

tkinter.Button(self, text='askdirectory', command=self.askdirectory).pack(\*\*button\_opt)

# define options for opening or saving a file

self.file\_opt = options = {}

options['defaultextension'] = '.txt'

options['filetypes'] = [('all files', '.\*'), ('text files', '.txt')]

options['initialdir'] = 'C:\\'

options['initialfile'] = 'myfile.txt'

options['parent'] = root

options['title'] = 'This is a title'

# This is only available on the Macintosh, and only when Navigation Services are installed.

#options['message'] = 'message'

# if you use the multiple file version of the module functions this option is set automatically.

#options['multiple'] = 1

# defining options for opening a directory

self.dir\_opt = options = {}

options['initialdir'] = 'C:\\'

options['mustexist'] = False

options['parent'] = root

options['title'] = 'This is a title'

def askopenfile(self):

"""Returns an opened file in read mode."""

return tkinter.filedialog.askopenfile(mode='r', \*\*self.file\_opt)

def askopenfilename(self):

"""Returns an opened file in read mode.

This time the dialog just returns a filename and the file is opened by your own code.

"""

# get filename

filename = tkinter.filedialog.askopenfilename(\*\*self.file\_opt)

# open file on your own

if filename:

return open(filename, 'r')

def asksaveasfile(self):

"""Returns an opened file in write mode."""

return tkinter.filedialog.asksaveasfile(mode='w', \*\*self.file\_opt)

def asksaveasfilename(self):

"""Returns an opened file in write mode.

This time the dialog just returns a filename and the file is opened by your own code.

"""

# get filename

filename = tkinter.filedialog.asksaveasfilename(\*\*self.file\_opt)

# open file on your own

if filename:

return open(filename, 'w')

def askdirectory(self):

"""Returns a selected directoryname."""

return tkinter.filedialog.askdirectory(\*\*self.dir\_opt)

if \_\_name\_\_=='\_\_main\_\_':

root = tkinter.Tk()

fileDialogExample(root).pack()

root.mainloop()

**Lab Task:**

1. Create file smstr7.

Append 10 students’ names with respective quality in the file. For example, qualities can be “hard-working”, “studious”, “genius”, “thoughtful”

1. Open the file smstr7 in read mode and print each line using a loop.
2. Open the file smstr7 in read mode and print each line using readline().
3. Create two files containing 2 lists of student names and cities. Append contents of the two files in each other so that at the end of your program the content of both files is written in both files.

**Bonus Lab Task:**

Notepad in python GUI

Lab 14

**Project Work:**

You were assigned projects on Machine Learning after the midterm exam.

**Lab Task:**

Complete your semester projects.