Appendix B

Python and SageMath

SageMath (originally Sage) is a computer algebra system that is built on top of Python, which is a popular general-purpose programming language. In this appendix we highlight a few features of Python through a series of SageMath cells. Pure Python code can generally be evaluated in these cells and most of what you see here is just Python. There are exceptions. For example, SageMath has enhanced capabilities to work with sets. In Python, the expression `set([0,1,2,3])` is a set of four integers, and certain basic set operations can be performed on these types of expressions. This is a valid expression in SageMath too, but a different SageMath expression, `Set([0,1,2,3])`, with a capital S, has enhanced properties. For example, we can create the power set of the SageMath expression, which we do in the discussion of iterators.

B.1 Python Iterators

All programming languages allow for looping. A common form of loop is one in which a series of instructions are executed for each value of some index variable, commonly for values between two integers. Python allows a bit more generality by having structures called "iterators" over which looping can be done. An iterator can be as simple as a list, such as `[0,1,2,3]`, but also can be a power set of a finite set, as we see below, or the keys in a dictionary, which is described in the next section.

B.1.1 Counting Subsets

Suppose we want to count the number of subsets of $\{0, 1, 2, ..., 9\}$ that contain no adjacent elements. First, we will define our universe and its power set. The plan will be to define a function that determines whether a subset is "valid" in the sense that it contains no adjacent elements. Then we will iterate over the subsets, counting the valid ones. We know that the number of all subsets will be 2 raised to the number of elements in U, which would be $2^{10} = 1024$, but let's check.

```
U=Set(range(10))
power_set=U.subsets()
len(power_set)
```

```
1024
```

The validity check in this case is very simple. For each element, k, of a set, B, we ask whether its successor, $k+1$, is also in the set. If we never get an answer of ”True” then we consider the set valid. This function could be edited to define validity in other ways to answer different counting questions. It's always a good idea to test your functions, so we try two tests, one with a valid set and one with an invalid one.

```
def valid(B):
    v=true
    for k in B:
        if k+1 in B:
            v=false
            break
    return v
[valid(Set([1,3,5,9])),valid(Set([1,2,4,9]))]
```

```
[True, False]
```

Finally we do the counting over our power set, incrementing the count variable with each valid set.

```
count=0
for B in power_set:
    if valid(B):
        count+=1
count
```

```
144
```

B.2 Dictionaries

B.2.1 Colors of Fruits

In Python and SageMath, a dictionary is a convenient data structure for establishing a relationship between sets of data. From the point of view of this text, we can think of a dictionary as a concrete realization of a relation between two sets or on a single set. A dictionary resembles a function in that there is a set of data values called the `keys`, and for each key, there is a `value`. The value associated with a key can be almost anything, but it is most commonly a list.

To illustrate the use of dictionaries, we will define a relationship between colors and fruits. The keys will be a set of colors and values associated with each color will be a list of fruits that can take on that color. We will demonstrate how to initialize the dictionary and how to add to it. The following series of assignments have no output, so we add a print statement to verify that this cell is completely evaluated.

```
fruit_color={}
fruit_color['Red']=['apple','pomegranate','blood␣orange']
fruit_color['Yellow']=['banana','apple','lemon']
fruit_color['Green']=['apple','pear','grape','lime']
fruit_color['Purple']=['plum','grape']
fruit_color['Orange']=['orange','pineapple']
print 'done'
```

We distinguish a color from a fruit by capitalizing colors but not fruit. The keys of this dictionary are the colors:

```
fruit_color.keys()
```

```
['Purple', 'Orange', 'Green', 'Yellow', 'Red']
```

As an afterthough, we might add the information that a raspberry is red as follows. You have to be careful in that if 'Red' isn't already in the dictionary, it doesn't have a value. This is why we need an if statement.

```
if 'Red' in fruit_color:
        fruit_color['Red']=fruit_color['Red']+['raspberry']
else:
        fruit_color['Red']=['raspberry']
fruit_color['Red']
```

```
['apple', 'pomegranate', 'blood_orange', 'raspberry',
    'raspberry']
```

A dictionary is iterable, with an iterator taking on values that are the keys. Here we iterate over the our dictionary to output lists consisting of a color followed by a list of fruits that come in that color.

```
for fruit in fruit_color:
    print [fruit,fruit_color[fruit]]
```

```
['Purple', ['plum', 'grape']]
['Orange', ['orange', 'pineapple']]
['Green', ['apple', 'pear', 'grape', 'lime']]
['Yellow', ['banana', 'apple', 'lemon']]
['Red', ['apple', 'pomegranate', 'blood_orange','raspberry']]
```

We can view a graph of this relation between colors and fruits, but the default view is a bit unconventional.

```
DiGraph(fruit_color).plot()
```

With a some additional coding we can line up the colors and fruits in their own column. First we set the positions of colors on the left with all x-coordinates equal to -5 using another dictionary called `vertex_pos`.

```
vertex_pos={}
k=0
for c in fruit_color.keys():
    vertex_pos[c]=(-5,k)
    k+=1
vertex_pos
```

```
{'Purple': (-5, 0), 'Orange': (-5, 1), 'Green': (-5, 2),
    'Red': (-5, 4), 'Yellow': (-5, 3)}
```

Next, we place the fruit vertices in another column with x-coordinates all equal to 5. In order to do this, we first collect all the fruit values into one set we call `fruits`.

```
fruits=Set([ ])
for v in fruit_color.values():
    fruits=fruits.union(Set(v))
k=0
for f in fruits:
    vertex_pos[f]=(5,k)
    k+=1
vertex_pos
```

```
{'blood_orange': (5, 0), 'grape': (5, 1), 'apple': (5, 2),
    'Purple': (-5, 0), 'plum': (5, 10), 'pomegranate': (5, 3),
    'pear': (5, 4), 'Yellow': (-5, 3), 'orange': (5, 7),
    'Green': (-5, 2), 'pineapple': (5, 6), 'Orange': (-5, 1),
    'lemon': (5, 8), 'raspberry': (5, 9), 'banana': (5, 5),
    'Red': (-5, 4), 'lime': (5, 11)}
```

Now the graph looks like a conventional graph for a relation between two different sets. Notice that it's not a function

```
DiGraph(fruit_color).plot(pos=vertex_pos,vertex_size=1)
```