Python and SAGE

- SAGE is a mathematical software
- SAGE is free software
- It is based on Python

Visit http://www.sagemath.org

- Python: interpreted, dynamically typed language
- Very convenient for many applications
- Slower than C, but many critical parts of SAGE are written in C, and interfaced with Python

Visit http://www.python.org

Data types

Integer

```
>> a = 5
>> b = -6
```

String

```
>> a = 'text'  # or "text"
>> len(a)
   4
>> a[0]
   't'
```

Rational

```
>> a = 1 / 2
>> b = 5 / 2  # Equal to 0 in Python, and 5 / 2 in SAGE
```

Floating-point

```
>> a = 1.0
>> b = 1.5 / 3
>> c = 2 * a * (3 + b) / 44 / 55
>> c = ((2 * a * (3 + b)) / 44) / 55
```

Tuples: ordered lists

Tuples are ordered and immutable

```
>> a = ( )          # Empty tuple
>> a = 1,           # One element tuple
>> a = (1)          # not a tuple!
>> a = (1, 2, 3)    # 3 element tuple
>> a = 1, 2, 3      # The same tuple
>> a[1]
   2
>> len(a)
   3
>> a[1] = 2
   TypeError: tuple does not support assignment!
>> sum(a), max(a), min(a)
   (6, 3, 1)
```

Data types: list

List: works as an array

```
>> a = [ ]        # Empty list
>> a = [ 1, 2 ] # Two elements in the list
>> a.append(3)
>> a
   [ 1, 2, 3 ]
>> len(a)
   3
>> a[0] = 5      # List becomes [ 5, 2, 3 ]
>> a[-1]
   3
>> a = 3 * [ 5 ]
   [ 5, 5, 5 ]
>> a = [ 5, ”teste” ]
>> a = [ 1, 2, 3 ]
>> sum(a), max(a), min(a)
   6, 3, 1
```

Data types: dictionary

Dictionary: efficient dictionary implementation

```
>> a = { }  # Empty dictionary
>> a = { "Alice": 1432, "Bob": 1717 }  # Social security numbers
>> "Alice" in a
   True
>> "Richard" in a
   False
>> a["Alice"]
   1432
>> a["Richard"]
   KeyError
>> a["Alice"] = 10
>> a["Richard"] = 5
>> a
   { "Alice": 10, "Bob": 1717, "Richard": 5 }
```

Conditionals

Some simple conditionals

```
a = { }
if not "Richard" in a:
    print "Richard does not have a number!"
    a["Richard"] = -1
elif a["Richard"] == -1:
    print "Richard didn't have a number last time!"
else:
    print "Richard's number is", a["Richard"]
```

Python blocks are given by indentation!

```
if a > b:
    do something
else:
    do something else
    if a > 2 * b:
        and yet something else
      oops! not good!
```

How would it look like in C?

```
if (a > b) {
    do something;
}
else {
    do something else;
    if (a > 2 * b) {
        and yet something else;
    }
}
```

Looping: while

While loop

```
while condition:
    do something
    break     # Finishes loop
    continue  # Skips to next iteration
else:  # optional!
    what is done when condition becomes False
    If exit through break, this is ignored!
```

Example: is a number prime?

```
f = 2
while f < a:
    if a % f == 0:
        print ”Number is not prime”
        break
    f += 1
else:
    print ”Number is prime”
```

Looping: for

Use for lists:

```
fruit_list = [ "orange", "grape", "banana" ]
for fruit in fruit_list:
    print fruit
else:
    print "End of list"
```

range:

```
>> range(a, b, s)
   [ a + ks : a + ks < b and k = 0, 1, 2, ... ]
>> range(3)   # The same as range(0, 3, 1)
   [ 0, 1, 2 ]
>> range(1, 3)
   [ 1, 2 ]
>> range(2, 2)
   [ ]
>> range(3, 0, -1)
   [ 3, 2, 1 ]
```

Looping: for

Looping with range:

```
total = 0
for x in range(10):
    total += x
# Now total = 0 + 1 + ... + 9 = 45
# The same as sum(range(10))
```

In C:

```
int total = 0;
for (int x = 0; x < 10; x++)
    total += x
```

When looping with `range`, use `xrange`

```
fac = 1
for x in xrange(1, 10):
    fac *= x

s = 1
for k in  range(10000000): s *= 1   # Takes 3.28s
for k in xrange(10000000): s *= 1   # Takes 2.73s
```

Looping through a dictionary

The dictionary is like a list of the keys!

```
a = { "Alice": 1, "Bob": 2 }
for name in a:
    print name, a[name]
```

Result:

```
Alice 1
Bob 2
```

Other lists associated with a dictionary:

```
>> a.keys()
   [ "Alice", "Bob" ]
>> a.values()
   [ 1, 2 ]
```

Functions

Function notation:

```
def func():  # No arguments!
    return "Hello"  # Return value is optional!

>> func()
   "Hello"
```

More complicated example:

```
def invert_if_bigger(a, b):
    if a > b: return b, a
    return a, b

>> invert_if_bigger(1, 2)
   (1, 2)
>> invert_if_bigger(2, 1)
   (1, 2)
```

Writing SAGE programs

- Write program to a file
  - Use a nice editor, line Emacs in Python mode!
  - This helps you with the indentation
- Load it in sage

Program "`fac.sage`"

```
def fac(p):
    x = 1
    for k in xrange(2, p + 1):
        x *= k
    return x

print fac(5)
```

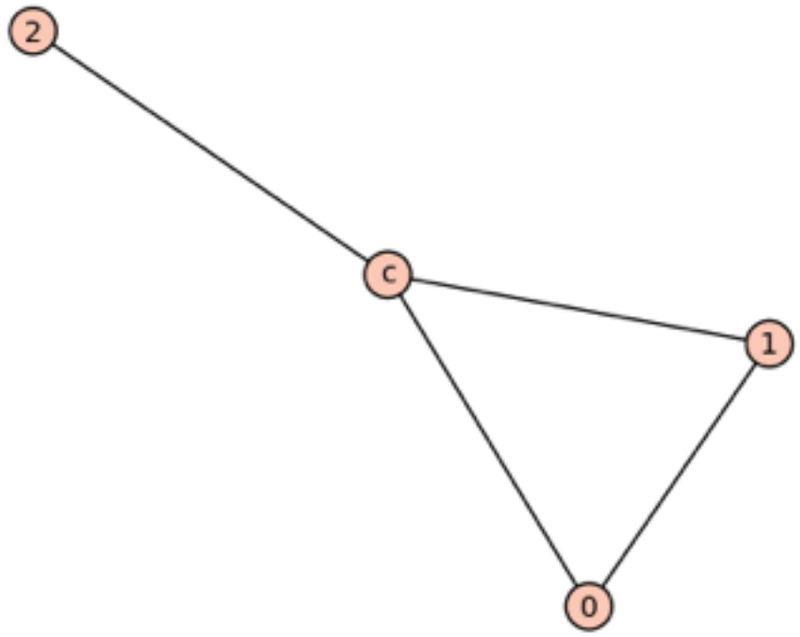
Sage console:

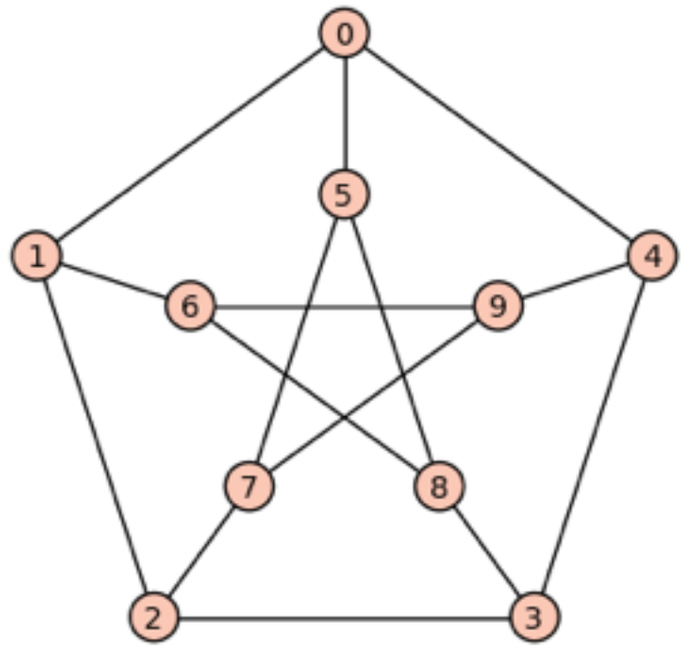
```
>> load("fac.sage")
   120
>> fac(6)
   720
```

SAGE Graph class

Used to represent undirected graphs:

```
>> a = { 0: [ 1, 'c' ], 'c': [ 1, 2 ] }  # Adjacency list
>> G = Graph(a)
>> G.plot()
>> G = graphs.PetersenGraph()
>> G.plot()
```

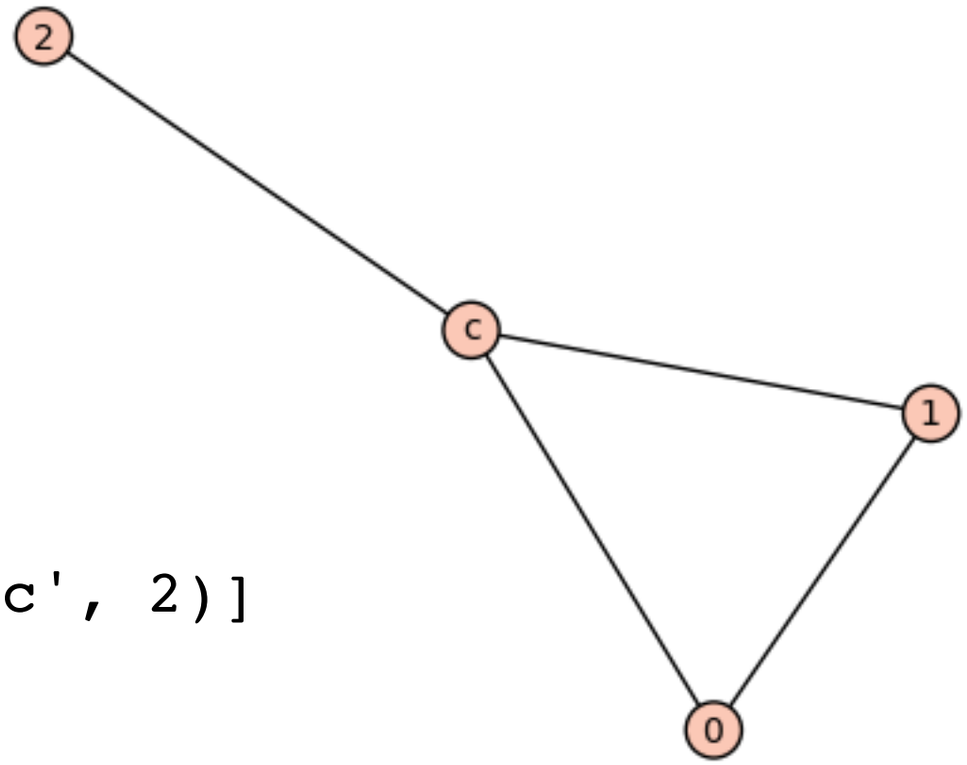




Iterating through graphs

List of vertices and the adjacency list of a vertex

```
>> a = { 0: [ 1, 'c' ], 'c': [ 1, 2 ] }
>> G = Graph(a)
>> G.vertices()
   [ 0, 1, 'c', 2 ]
>> G[1]
   [ 0, 'c' ]
>> G['c']
   [ 0, 1, 2 ]
>> G.edges(labels = False)
   [(0, 1), (0, 'c'), (1, 'c'), ('c', 2)]
```



How to iterate:

```
for u in G:  # You don't need to use G.vertices() here!
    print 'Here are the neighbors of', u
    for v in G[u]:
        print v
```