Hello, world!

In [2]:
```
print "Hello, world!"
```

```
Hello, world!
```

In [3]:
```
def sayhello():
    print "Hello, world!"
```

In [4]:
```
sayhello()
```

```
Hello, world!
```

In [5]:
```
def sayhello(name):
    print "Hello, " + name
```

In [6]:
```
sayhello('Rick')
```

```
Hello, Rick
```

Exercise: Write a “Hello, world” program that uses a function with a parameter to greet you by name

Basic types

In [7]:
```
1 # integer
```

Out[7]: 1

In [8]:
```
3.14 # floating-point
```

Out[8]: 3.14

In [9]:
```
3.14e17 # floating-point (scientific notation)
```

Out[9]: 3.14e+17

In [10]:
```
123456789123456789123456789 # long integer (unbounded)
```

Out[10]: 123456789123456789123456789L

In [11]:
```
# Type introspection
print type(4)
print type(3.14)
```

```
<type 'int'>
<type 'float'>
```

In [12]:
```
1 + 1j
```

Out[12]: (1+1j)

In [13]:
```
1j * 1j
```

Out[13]: (-1+0j)

Variables and basic arithmetic

In [14]:
```
x = 1
x
```

Out[14]: 1

In [15]:
```
x + 1
```

Out[15]: 2

In [16]:
```
y = x * 5
y
```

Out[16]: 5

In [17]:
```
y *= 3 # in-place assigment
print y
```

```
15
```

In [18]:
```
y % 2 # modulo arithmetic
```

Out[18]: 1

In [19]:
```
y / 2 # result dependent on Python version
```

Out[19]: 7

In [20]:
```
y / 2.0 # force floating point
```

Out[20]: 7.5

In [21]:
```
y // 2.0 # force integer (floor) division)
```

Out[21]: 7.0

In [22]:
```
2==2
```

Out[22]: True

In [23]:
```
2 <= 1 # comparisons are < > <= >= == !=
```

Out[23]: False

In [84]:
```
2 != 3
```

Out[84]: True

In [85]:
```
# exponentiation
2 ** 4
```

Out[85]: 16

In [25]:
```
True and False
```

Out[25]: False

In [26]:
```
True or False
```

Out[26]: True

In [27]:
```
True and not False
```

Out[27]: True

Strings

In [28]:
```
print 'This is a string with single quotes'
print "This is a string with double quotes"
```

```
This is a string with single quotes
This is a string with double quotes
```

In [29]:
```
print "We can embed 'single quotes' within double quotes without escaping"
print 'We can also embed "double quotes" within single quotes without escaping'
print "But we must escape \"double quotes\" inside double quotes and single quotes inside singl
```

```
We can embed 'single quotes' within double quotes without escaping
We can also embed "double quotes" within single quotes without escaping
But we must escape "double quotes" inside double quotes and single quotes inside single quotes
```

In [30]:
```
print "Strings can be" + " concatenated together " + "using the '+' operator"
```

```
Strings can be concatenated together using the '+' operator
```

In [31]:
```
print '''We can also begin a string with three single-quote characters
or three double-quote characters. This allows us to build multi-line
strings without using the \\n escape sequence.'''
```

```
We can also begin a string with three single-quote characters
or three double-quote characters. This allows us to build multi-line
strings without using the \n escape sequence.
```

In [32]:
```
print u'Unicode strings have a "u" prefix.'
print u'You can use encoded unicode characters in your source code: 海淀区科学院南路2号融科资讯中心C
```

```
Unicode strings have a "u" prefix.
You can use encoded unicode characters in your source code: 海淀区科学院南路2号融科资讯中心C座南楼8层
```

We *can* use escape sequences inside strings, however. Escape sequences start with a backslash character (). Some common escape sequences appear below:

- \n - newline
- \r - carriage return
- \t - tab character
- \ - literal backslash

- \x20 - hex escape (in this case, ASCII character 0x20, the space)
- \u4e0b - unicode escape (hex sequence)

You can index into a string using either a single integer or a "slice." The result is always another string (there is no 'char' type in Python).

In [33]:
```
s = 'The quick brown fox'
s[0]
```

Out[33]: 'T'

In [34]:
```
type(s[0])
```

Out[34]: str

In [35]:
```
s[1:15]
```

Out[35]: 'he quick brown'

In [36]:
```
s[1:15:2] # 'step' by 2
```

Out[36]: 'h uc rw'

In [37]:
```
s[:3] # beginning of range omitted
```

Out[37]: 'The'

In [38]:
```
s[3:] # end of range omitted
```

Out[38]: ' quick brown fox'

In [39]:
```
s[-1] # negative indexing from the end
```

Out[39]: 'x'

In [40]:
```
s[-3:] # last 3 characters
```

Out[40]: 'fox'

In [41]:
```
s[::-1] # negative step reverses the sequence
```

Out[41]: 'xof nworb kciuq ehT'

In [86]:
```
len(s) # length of string
```

Out[86]: 19

In [89]:
```
x = u'海'
print len(x) # unicode length
```

```
1
```

In [94]:
```
y = x.encode('utf-8') # encode to bytes ('str')
len(y)
```

Out[94]: 3

In [96]:
```
# decode utf-8 back to unicode
print unicode(y, 'utf-8')
```

```
海
```

Lists

Lists are mutable, dynamically typed sequences, and they are used frequently in Python.

In [42]:
```
# This is a list
print [ 1,2,3]
```

```
[1, 2, 3]
```

In [43]:
```
# We can append to lists
lst = [ 1, 2, 3 ]
lst.append(4)
print lst
```

```
[1, 2, 3, 4]
```

In [44]:
```
# We can index into lists (0-based indexing)
print lst[2]
```

```
3
```

In [45]:
```
# We can insert, remove, and update
lst.insert(1, 42)
print lst
```

```
[1, 42, 2, 3, 4]
```

In [46]:
```
lst.remove(2)
print lst
```

```
[1, 42, 3, 4]
```

In [47]:
```
del lst[2]
print lst
```

```
[1, 42, 4]
```

In [48]:
```
lst[1] = 43
print lst
```

```
[1, 43, 4]
```

In [49]:
```
# pop() removes and returns the last element
print lst.pop()
```

```
4
```

In [50]:
```
print lst
```

```
[1, 43]
```

In [51]:
```
# Lists are dynamically typed
lst = [ 'This', 'is', 'a', 'list', 'with', 7, 'elements' ]
lst
```

Out[51]:
```
['This', 'is', 'a', 'list', 'with', 7, 'elements']
```

In [52]:
```
# Lists can be sliced just like strings
lst[::-2]
```

Out[52]:
```
['elements', 'with', 'a', 'This']
```

Tuples

Tuples are *immutable* sequences.

In [53]:
```
# this is a tuple
print (1, 2, 3)
```

```
(1, 2, 3)
```

In [54]:
```
print ('Tuples', 'are', 'also', 'dynamically', 'typed', 42)
```

```
('Tuples', 'are', 'also', 'dynamically', 'typed', 42)
```

In [55]:
```
# Tuples can be indexed
t = (1,2,3)
print t[0]
```

```
1
```

In [56]:
```
# Tuples can be "unpacked"
x,y = (1,2)
print x
print y
```

```
1
2
```

Basic control structures

In [57]:
```
if True:
    print 'This is true'
else:
    print 'This is false'
```

```
This is true
```

In [58]:
```
x = 0
while x < 5:
    x = x + 1
    print x
```

```
1
2
3
4
5
```

In [59]:
```
lst = [ 1, 2, 3 ]
for x in lst:
    print x
```

```
1
2
3
```

In [60]:
```
print range(3)
for x in range(3):
    print x
```

```
[0, 1, 2]
0
1
2
```

In [61]:
```
print xrange(3)
for x in xrange(3):
    print x
```

```
xrange(3)
0
1
2
```

In [62]:
```
for x in xrange(10):
    if x % 2 == 0:
        continue
    if x > 5: break
    print x
```

```
1
3
5
```

Exercises

- Write a function that sums the values in a list using a `for` loop
- Write a function that sums the even-numbered values in a list
- Write a function that returns the reversed version of a list

Dicts

A *dict* is a hash table (also known as a "dictionary"). Dicts are pervasive in Python.

In [63]:
```
print { 'key': 'value' }
```

```
{'key': 'value'}
```

In [64]:
```
d = { 'key1': 1, 'key2': 'foo' }
print d
```

```
{'key2': 'foo', 'key1': 1}
```

In [65]:
```
d['key1']
```

Out[65]:
```
1
```

In [66]:
```
d['key3'] = 'bar'
```

In [67]:
```
print d
```

```
{'key3': 'bar', 'key2': 'foo', 'key1': 1}
```

In [68]:
```
# dicts are unordered, but we can get a list of their keys,
# values, or (key,value) pairs
print d.keys()
```

```
['key3', 'key2', 'key1']
```

In [69]:
```
print d.values()
```

```
['bar', 'foo', 1]
```

In [70]:
```
print d.items()
```

```
[('key3', 'bar'), ('key2', 'foo'), ('key1', 1)]
```

In [71]:
```
# dict keys can be any *immutable* type in Python
d = { 'foo': 1, 2: 'bar' }
print d
```

```
{2: 'bar', 'foo': 1}
```

In [72]:
```
d[(1,2)] = 'baz'
print d
```

```
{(1, 2): 'baz', 2: 'bar', 'foo': 1}
```

Items can be removed using the `del` keyword

In [73]:
```
del d[2]
print d
```

```
{(1, 2): 'baz', 'foo': 1}
```

A `dict` can be iterated through in a space-efficient using `iterkeys`, `itervalues`, and `iteritems`:

In [74]:
```
for k in d.iterkeys():
    print k
```

```
(1, 2)
foo
```

In [75]:
```
for v in d.itervalues():
    print v
```

```
baz
1
```

In [76]:
```
for k,v in d.iteritems():
    print k, v
```

```
(1, 2) baz
foo 1
```

In [77]:
```
'foo' in d # Test for key membership
```

Out[77]: True

Exceptions

Python handles errors by throwing *exceptions*. For instance, trying to read a non-existent key in a dict:

In [78]:
```
d['does not exist']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/vagrant/<ipython-input-78-17d039c47522> in <module>()
----> 1 d['does not exist']

KeyError: 'does not exist'
```

To handle exceptions gracefully, we must enclose them in a `try:` *block*:

In [79]:
```
try:
    x = d['does not exist']
    print 'This statement never executes!'
except KeyError:
    print 'There was a key error!'
```

```
There was a key error!
```

We can also write code that will *always* run, whether an exception is raised or not:

In [80]:
```
try:
    x = d['does not exist']
except KeyError:
    print 'There was a key error!'
finally:
    print 'This always runs!'
```

```
There was a key error!
This always runs!
```

In [81]:
```
try:
    x = d['key1']
except KeyError:
    print 'There was a key error!'
finally:
    print 'This always runs!'
```

```
There was a key error!
This always runs!
```

If we want code that only runs when there is *not* an error, we can use the `else:` clause:

In [82]:
```
try:
    x = d['key1']
except KeyError:
    print 'There was a key error!'
else:
    print 'The try: block completed without error.'
finally:
    print 'This always runs!'
```

```
There was a key error!
This always runs!
```

To *cause* an exception, use the `raise` keyword:

In [83]:
```
raise KeyError('This is a key error')
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/vagrant/<ipython-input-83-04dfd7050815> in <module>()
----> 1 raise KeyError('This is a key error')

KeyError: 'This is a key error'
```

Exercises

- Write a collection of functions to manage a telephone directory. The directory should be stored in a dict and passed as the first argument to each function. These functions should include `add_number(directory, name, number)`, `remove_number(directory, name)`, and `lookup_number(directory, name)`.
- Update your function so that `remove_number` does not raise an exception when you remove a non-existent entry.