Base Numbering Systems

- Numbers can be represented by and to the computer using various base number systems.
- At this point, we are particularly interested in:
  - Binary (base 2)
  - Octal (base 8) # Not in lab 4, but common
  - Decimal (base 10)
  - Hexadecimal (base 16)

Base Numbering Systems

- Python has various mechanisms for handling different number bases:
  - Base 10 is recognized implicitly:
    ```python
    >>> 10
    10
    >>> 105
    105
    >>> 2112
    2112
    >>>
    ```
Base Numbering Systems

- Others are not:

```python
>>> 011
>>> 0f5
Traceback (most recent call last):
  File '<pyshell#18>', line 1, in <module>
    0f5
NameError: name '0f5' is not defined
>>> |
```

- Numbers preceded by a zero are interpreted by Python as octal numbers
  - 013 = (1 * 8¹) + (3 * 8⁰) = 8 + 3 = 11

- Numbers preceded by a zero and the character x are interpreted by Python as hexadecimal numbers
  - 0x1B = (1 * 16¹) + (11 * 16⁰) = 16 + 11 = 27

- There are built-in functions that can handle some transitions:
  - `eval` – convert a string to number (string can be hex or octal number)
  - `hex(<number>)` – convert to hex string
  - `oct(<number>)` – convert to octal string
  - `bin(<number>)` – convert to binary string

- Binary: digits 0-1
- Octal: digits 0-7
- Decimal: digits 0-9
- Hexadecimal: digits 0-9 & A(10)-F(15)
String as a Sequence

- **String**: An immutable sequence of characters
  - **immutable**: cannot be changed
  - **sequence**: a particular order in which things follow each other
    - forward index: \(0\) through \(n-1\)
    - backward index: \(-1\) through \(-n\)
  - **character**: individual ascii symbols

Indexing example

```python
>>> greet = "Hello Bob"
>>> print greet[0], greet[2], greet[4]
H l o
>>> x = 8
>>> print greet[x - 2]
B
```

Indexing example - from the right

- In a string of \(n\) characters, the last character is at position \(n-1\) since we start counting with \(0\).
- We can index from the right side using negative indexes.

```python
>>> print greet[-1], greet[-3]
b o
```
String Data Structure

- Immutability:
  - individual elements (characters) can not be changed once created
  - the string can be recreated
  - the variable can be re-defined

String Immutability

- An attempted String mutation

String re-creation

- Recreating or reassigning a string is fine:

String Methods

- Many string methods return a new string (because they cannot modify (mutate) the original string).
  - aStr = "hello world"
  - bStr = aStr.capitalize() # Does this change aStr?
  - aStr = aStr.capitalize() # Is this legal?
Sequence Operators

3.6 Sequence Types — str, unicode, list, tuple, buffer, xrange

There are ten sequence types: string, Unicode string, list, tuple, buffer, and range objects.

You've already seen these Sequence Operations. What about a substring?

**Slicing a string**

- Slicing: `<string>[<start>:<end>]`
- start and end must both be ints
- The slice contains the substring beginning at position start and runs up to but doesn’t include the position end.

**Slicing Example**

```
>>> greet[0:3]
'Hel'
>>> greet[5:9]
' Bob'
>>> greet[:5]
'Hello'
>>> greet[5:]
' Bob'
>>> greet[:]
'Hello Bob'
```

Hint: When slicing it helps to think of the slice indexes between the characters, then 0:3 is very clear.
String Declaration & Initialization

• Declaring an Empty String
  – `a_string = ''` empty single or double quotes

• Note: nothing inherently special about name (just another identifier) so self-documenting code helps...
  – `x = 5`            `x = ''`          `first_name = ''`

MIN & MAX Functions

• `min(sequence)`: returns the element in the sequence that has the minimum “value”

• `max(sequence)`: returns the element in the sequence that has the maximum “value”

• Based on ASCII code value for string sequences

ORD & CHR Functions

• `ord(char)`: converts single character to corresponding ASCII integer value

• `chr(int)`: converts integer value to corresponding character symbol

• Based on ASCII code value
  – American Standard Code for Information Interchange
  – 7 binary bits ⇒ 128 unique symbols

• Python also supports Unicode (16 bits)

ASCII Table

American Standard Code for Information Interchange
ORD & CHR Functions

```
>>> ord('A')

32
>>> chr(65)

'A'
>>> ord('AA')

Is it alphabetical ordering? Be careful: min('abcWXY')?
```

MIN & MAX Functions

```
>>> min('123abc')

1
>>> max('123abc')

'c'
>>> min('123oHa')

'1'
>>> max('123oHa')

'H'

```

String Methods

```
>>> a_string = 'this is a sentence.'
>>> new_string = a_string.capitalize()
>>> new_string

'This is a sentence.'
>>> a_string

'this is a sentence.'
```
Comparison Operations

Sequence Comparison Operations

Dissecting Data Streams
Dissecting Data Streams

def main():
    test_str = "FAC50000BC4A01015CC01010"
    print "Address Data"
    print "------------"
    print "  " + test_str[0:4] + "  " + test_str[4:8]
    print "  " + test_str[8:12] + "  " + test_str[12:16]
    print "  " + test_str[16:20] + "  " + test_str[20:24]
main()

Terminology / Concepts

• Binary Number System
• Octal Number System
• Hexadecimal Number System
• ASCII
• Slicing/Substring

Tuple and Lists Data Structures

• Tuple: An immutable sequence of valid Python data types
• List: A mutable sequence of valid Python data types
• Tuples, Lists and Strings are all Python sequence data types.

Tuple Declaration & Initialization

>>> animal_tup = ()
>>> type(animal_tup)
<type 'tuple'>
>>> animal_tups = (1, 7, 3, -19)
>>> type(animal_tups)
<type 'tuple'>
>>> animal_age = [34]
>>> type(animal_age)
<type 'list'>
>>> animal_age = [34, ] # Gotta have the comma
>>> type(animal_age)
<type 'list'>

Oops... what happened?

• Note: nothing inherently special about name (just another identifier) so self-documenting code helps...
  - x = 5  x = 1  employee_tuple = ()

Empty tuple
Tupple with data
Not a tuple
Tuples are Sequences

```python
def tupleSequence():
    aTuple = [34, 45, 87, 99]
    print("(min),", aTuple)
    bTuple = 12, 13, 14 # Concatenation

    print("Tuple has 34!", 34 in aTuple)
    print("Tuple value 2!", aTuple[2])
    print("Tuple slice!", aTuple[2:])

>>> aTuple = [12, 23, 34]
>>> bTuple
(12, 23, 34)
>>> aTuple[2] = 22
Traceback (most recent call last):
  File "sample code", line 1, in _
    aTuple[2] = 22
TypeError: tuples object does not support item assignment
```

Still immutable though

Tuples can contain multiple data types

```python
>>> aTuple = ('Dan', 'was', 25, 300.0, 'seconds')
>>> bTuple = ('Something', 35, aTuple)
```

Tuples can even contain other tuples!

```python
>>> print(aTuple)
('Dan', 'was', 25, 300.0, 'seconds')
>>> print(bTuple)
('Something', 35, ('Dan', 'was', 25, 300.0, 'seconds'))
>>> 
```

Tuples as Lookup Tables

- Tuples are frequently used to created lookup tables.

- Requirement: Ask the user for a number and convert that to an appropriate month

- Lets try it!

Lists as also sequences

```python
>>> aList = []
>>> type(aList)
<type 'list'>
```

```python
>>> gradeList = [89, 98, 101, 23]
>>> type(gradeList)
<type 'list'>
```

```python
>>> gradeList = [89]
>>> type(gradeList)
<type 'list'>
```

- Lists are exactly like tuples, EXCEPT they are mutable!
Lists: Mutable!

```python
>>> gradeList = [89, 98, 101, 23]
```

Whoa there cowboy…

- Shouldn’t there be some special operators that work with mutable data structures like lists?
  - Yep --- wait till next week!
  - http://docs.python.org/library/stdtypes.html#mutable-sequence-types

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+</code></td>
<td>concatenates two <code>a</code> objects</td>
</tr>
<tr>
<td><code>+</code></td>
<td>shallow copies of <code>a</code> concatenated</td>
</tr>
<tr>
<td><code>+</code></td>
<td>i-th item of <code>a</code> or <code>b</code></td>
</tr>
<tr>
<td><code>+</code></td>
<td>slice of <code>a</code> from <code>i</code> to <code>j</code></td>
</tr>
<tr>
<td><code>+</code></td>
<td>slice of <code>a</code> from <code>i</code> to <code>j</code> with step <code>k</code></td>
</tr>
<tr>
<td><code>+</code></td>
<td>length of <code>a</code></td>
</tr>
<tr>
<td><code>+</code></td>
<td>largest item of <code>a</code></td>
</tr>
</tbody>
</table>

Note on Tuple Initialization

```python
(a_tuple = |)
print a_tuple
print tuple_a
print tuple_b
main()
```
Accessing Tuple Elements

```
# tuples.py - /users/rheshma/python/tuples.py

def main():
    a_tuple = (23, 45, 'box', 12.5, (1,2,3))
    print a_tuple[1], '+', a_tuple[2], '+', a_tuple[4]
    print a_tuple[2][0], '*', a_tuple[4][2]

    # Modify tuple elements
    a_tuple[2][0] = 5
    print a_tuple

main()
```

Repetition & Concatenation

```
# tuples.py - /users/rheshma/python/tuples.py

a_tuple = (2, 'box', 12.5, (1,2,3))
print a_tuple
a_tuple += 2
print a_tuple
a_tuple += (5,6)
print a_tuple
```

Modifying Tuple Elements

```
# tuples.py - /users/rheshma/112/tuples.py

def main():
    a_list = [1,2,3]
    a_tuple = ('box', 12.5, a_list)
    print a_tuple
    a_list[1] = 5
    print a_tuple  # While a tuple may be immutable, tuple elements may contain embedded references to mutable data types

main()
```

Practice with slicing/indexing

- Slicing and indexing is critical to know
- aString = 'abc123'
- aString[2]
- aString[0:4]
- aString[-2]
- aString[3:]

```
(aString[2] == 'b')
```
**Practice with slicing/indexing**

- Slicing and indexing is critical to know
- `a= ['abc', 123, 'ddd', 999]`
- `a[2]`
- `a[0:4]`
- `a[-2]`
- `a[1:]`

---

**String Justification**

- Justification
  - aligns text within a text field in a particular fashion:
    - left justified - "text"
    - center justified - "text"
    - right justified - "text"
  - `print "text".ljust(20)`
  - `print "text".center(20)`
  - `print "text".rjust(20)`

---

**Formatted Printing**

- `print "abc", 123, 'ddd', 999`
- `print "abc", 123, 'ddd', 999 x = 1234.567890`
- `print "abc", 123, 'ddd', 999 print "abc", 1234.567890`

---

**Terminology / Concepts**

- Indefinite Loop
- Definite Loop
- Mutable / Immutable
- Sequence
- String Data Type
- Tuple Data Type
- Justification