Slicing

- A slice is a subset of a sequence L, and is also a sequence.
- Simple slice is a range of elements:
  
  ```python
  s = "abcdefg"
  s[1:4] == "bcd"
  ```
- Note that all Python ranges exclude the last index.
- The generic form of slice is `sequence[first:last:step]`.
- If any of the three parameters are left out, they are set to default values as follows: `first=0`, `last=len(L)`, `step=1`.
- So, for instance "abcde"[1:] == "bcde"
- The step parameter selects elements of range that are step distance apart from each other.
Modifying sequences

▶ We can assign values to elements of a list by indexing or by slicing

▶ Example: Let's bind L to the list [11, 13, 20, 32]. Now we can assign L[1] = 2, which changes the list to [11, 2, 20, 32].

▶ Or we can assign a list to a slice: L[1:3] = [4] to change the list to [11, 4, 32]

▶ We can also modify a list by using the *mutating methods* of the list class, namely the methods: append, extend, insert, remove, pop, reverse, sort

▶ Note that we cannot perform these modifications on tuples or strings since they are *immutable*
Generating sequences

- Trivial lists can be tedious to write: \([0,1,2,3,4,5,6]\)
- The function `range` creates lists automatically. The above list can be created with the function call `range(0,7)`
- Note again that the end value is not included to the list
- The range function works in similar fashion to slices. So, for instance the step of the sequence can be given: `range(0,7,2)` will produce the list \([0,2,4,6]\)
A **list comprehension** is an expression that allows creating a more complicated lists.

This method is familiar from mathematics:

\[ \{ a^3 : a \in \{1, 2, \ldots, 10\} \} \]

The same can be written in Python as a list comprehension:

```python
>>> [ i**3 for i in range(1,11)]
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```
The generic form of a list comprehension is:

\[
[\text{expression for element in iterable lc-clauses}]
\]

The iterable can be any sequence, or something more general.

The \textit{lc-clauses} consists of zero or more of the following clauses:

\begin{itemize}
  \item for \textit{elem} in \textit{iterable}
  \item if \textit{expression}
\end{itemize}
A more complex example:

```
print [ 100*a + 10*b +c for a in range(0,10)
       for b in range(0,10)
       for c in range(0,10) ]
```

What will this expression print?
Using iterators with for loops 1

- If a list is only needed so that a for loop can iterate through its elements, a better option is available.
- A long list can consume lots of memory. Using an iterator allows us to use only a moderate amount of memory.
- An iterator gives an item at a time. The elements don’t need to exist in physical memory simultaneously, but can be generated as needed.
- The for loop can therefore at each round ask for a new element from the iterator.
Using iterators with for loops

- If you need to iterate over a range of numbers, use `xrange` instead of a range.
- The `xrange` function works in the same way as `range`, but it doesn't return a list but instead an iterator.
- Example:
  ```python
  for i in xrange(0,11):
    print i
  ```
We can do similar transform for list comprehensions as we did for ranges

*Generator expressions* are exactly like list comprehensions except they are bounded by parentheses () instead of brackets []

A generator expression will result in an iterator instead of a list
Dictionaries 1

- A dictionary is a dynamic, unordered container
- Instead of using integers to access the container’s elements, the dictionary uses keys to access the stored values
- The dictionary can be created by listing the comma separated key-value pairs in braces. Keys and values are separated by a colon. A tuple (key, value) is called an item of the dictionary.
- An example of dictionary creation and usage:

  ```python
  >>> d={"key1":"value1", "key2":"value2"}
  >>> print d["key1"]
  "value1"
  >>> print d["key2"]
  "value2"
  ```
Keys can have different types even in the same container. So the following code is legal:

d={1: "a", "z": 2}

The only restriction is that the keys must be hashable. That is, there has to be a mapping from keys to integers.

Alternative ways of creating a dictionary

- `dict(key1="value1", ..., keyn="valuen")`
- `dict([("key1","value1"), ..., ("keyn","valuen")])`

If a key is not found in a dictionary the indexing `d[key]` results in an error (`exception KeyError`)

But an assignment with nonexisting key causes the key to be added in the dictionary associated with the corresponding value.
Nonmutating dictionary methods

- D.copy()
- D.has_key(k)
- D.items()
- D.keys()
- D.values()
- D.iteritems()
- D.iterkeys()
- D.itervalues()
- D.get(k[,x])
Mutating dictionary methods

- D.clear()
- D.update(d1)
- D.setdefault(k[,x])
- D.pop(k[,x])
- D.popitem()
A set is a dynamic, unordered container

It works a bit like dictionary but only keys are stored. And each key can be stored only once

The set requires that the values to be stored are hashable

An empty set can be created with the call `set()`, or a set can be constructed from an existing sequence: `set([1,2,10,'a'])`
Nonmutating set methods

- `s.copy()`
- `s.difference(s1)`
- `s.intersection(s1)`
- `s.issubset(s1)`
- `s.issuperset(s1)`
- `s.symmetric_difference(s1)`
- `s.union(s1)`
Mutating set methods

- s.add(x)
- s.clear()
- s.discard()
- s.pop()
- s.remove(x)

The nonmutating methods difference, intersection, symmetric_difference, and union have corresponding mutating methods: difference_update, intersection_update, symmetric_difference_update, and update
Operator forms of set methods

- The methods difference, intersection, symmetric_difference and union can be represented conveniently by operators: $s - s1$, $s & s1$, $s ^\text{sym} s1$, and $s | s1$

- The corresponding mutating methods have augmented assignment forms: $s-=s1$, $s&=s1$, $s^\text{sym}=s1$, and $s|=s1$
Membership testing and unpacking of containers

To find out whether a container includes an element, the `in` operator can be used. The operator returns a truth value.

Examples: `1 in [1,2]`, `key in dict(...)`, `element in set`, `'q' in "abcde"`

*Unpacking assignment* extracts the elements of a container into variables.

Examples:

- `first, second = (1,2)`
- `a,b,c = "hei"`
- `d=dict(...)`
  ```
  for key,value in d.items(): ...
  ```

In membership testing and in unpacking only the keys of a dictionary are used, unless either `values` or `items` (like above) are explicitly asked.
Deleting bindings, items and attributes

- To remove the binding of a variable, use the `del` statement.

- Example:
  ```python
  s="hello"
  del s
  ```

- To delete an item of a **container** the `del` statement can again be applied:
  ```python
  L=[13,23,40,100]
  del L[1]
  L==[13,40,100]
  ```

- In similar fashion `del` can be used to delete a slice.

- Later we will see that `del` can also delete attributes of an object.
Printing a complicated string, if we use several expressions for the print statement, can be quite clumsy

For example:

```python
a = 3; b = 6
print "The sum", a,"+",b,"equals", a+b
```

Also, the print statement separates the printed expressions by a space, which might not always what we want.
Another way to print a complicated string is to use the so-called *string-formatting expression*: `format % values`

The format string tells us where in the string we want some values to be placed by using a special notation.

And the values to be placed in the string are listed in the tuple `values`.

If only a single value is used in the format string, the value doesn’t have to be in a tuple, but can also be used without parentheses.
The example on the previous page can be done with the string-formatting expression as follows:

```
a = 3; b = 6
print "The sum %i + %i equals %i" % (a, b, a + b)
```

The resulting string is "The sum 3 + 6 equals 9"

The special notation %i will be replaced by integers that are given in the tuple after the % operator

There are similar special notations for other basic types besides integers
Format specifier syntax

The special syntax consists of the following elements, in order:

- The specifier starts with % character
- Optional element name in parentheses
- Zero or more flags
- Optional minimum width of the field, or a * in case you want to give the width in the values tuple
- Optionally, a dot followed by the precision, or a dot followed a * in case you want to give the precision in the values tuple
- A conversion character that denotes the printing type
## String-formatting conversion characters

<table>
<thead>
<tr>
<th>Format</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>d,i</code></td>
<td>signed decimal integer</td>
</tr>
<tr>
<td><code>u</code></td>
<td>unsigned decimal integer</td>
</tr>
<tr>
<td><code>o</code></td>
<td>unsigned octal integer</td>
</tr>
<tr>
<td><code>x,X</code></td>
<td>unsigned hexadecimal integer: lowercase, uppercase</td>
</tr>
<tr>
<td><code>e,E</code></td>
<td>float in exponential form: lowercase e, uppercase E</td>
</tr>
<tr>
<td><code>f,F</code></td>
<td>float in decimal form</td>
</tr>
<tr>
<td><code>g,G</code></td>
<td>like e/E when exponent &gt;= 4 or exponent &lt; precision, otherwise like f,F</td>
</tr>
<tr>
<td><code>c</code></td>
<td>single character</td>
</tr>
<tr>
<td><code>r</code></td>
<td>string, formed using <code>repr</code> function</td>
</tr>
<tr>
<td><code>s</code></td>
<td>string, formed using <code>str</code> function</td>
</tr>
<tr>
<td><code>%</code></td>
<td>literal % character</td>
</tr>
</tbody>
</table>
The optional conversion flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>The conversion uses an alternative form</td>
</tr>
<tr>
<td>0</td>
<td>Pad with zeros instead of spaces</td>
</tr>
<tr>
<td>-</td>
<td>Conversion is left justified</td>
</tr>
<tr>
<td>space</td>
<td>A space is placed before a positive number</td>
</tr>
<tr>
<td>+</td>
<td>Always use either + or - in front of a number</td>
</tr>
</tbody>
</table>