Sequence types

- `str` and `bytes` are *sequence types*
- Sequence types have several operations defined for them

Indexing

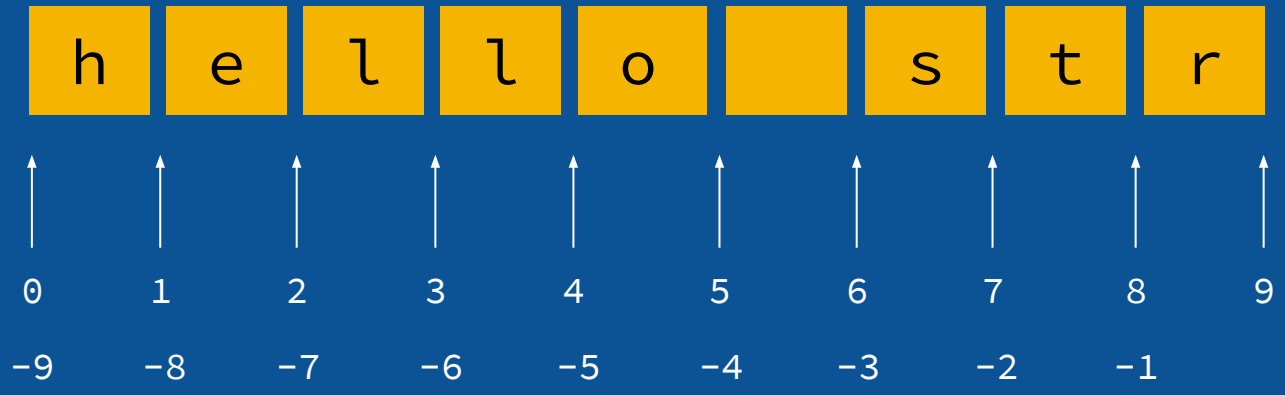
- Each element in a sequence can be extracted by using indexing syntax
  `seqtype[<integer>]`
- First element has index `0`

```
>>> mystr = 'hello str'
>>> a = mystr[0]
>>> print(a)
h
```

Python

Indexing

- Indexes can be thought of as *positions* rather than elements
  Makes slicing syntax more intuitive



Sequence Types: `list` and `tuple`

- While `str` only contains a sequence of characters
- `list` and `tuple` are containers for *any* type or collection of types
- `list` literal enclosed in `[ ]`
  Elements separated by commas
- `tuple` literal usually enclosed in `( )`
  Elements separated by commas
  Parenthesis mandatory for empty tuple

```
>>> mytuple = ('string', 83, 1+4j)
>>> print(type(mytuple))
<class 'tuple'>

>>> mylist = ['lststring', 12.22, b'\xAF']
>>> print(type(mylist))
<class 'list'>
```

Sequence Types: `list` and `tuple`

- Support the same indexing operations as `str` and `bytes`

```
>>> print(mytuple[2])
(1+4j)
>>> print(mylist[0])
lststring
```

Slicing

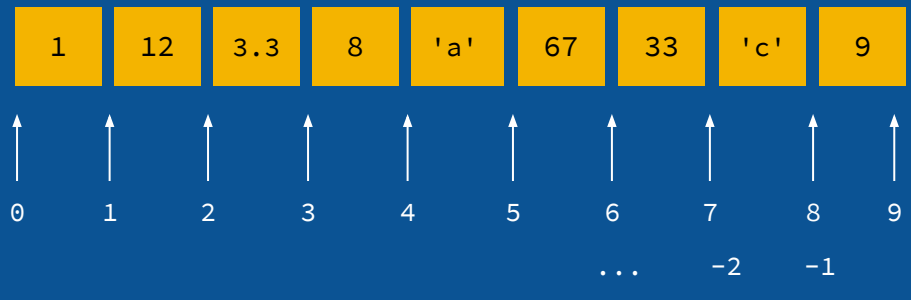
- Allows us to obtain a subset of the sequence
  `seqtype[start:stop:step]`
- Omitted indexes take default values
  start: `0`
  stop: `<last index>`
  step: `1`

Slicing Examples

```
mylst = [1,12,3.3,8,'a',67,33,'c',9]

>>> print(mylst[1:4])
[12, 3.3, 8]
>>> print(mylst[1:])
[12, 3.3, 8, 'a', 67, 33, 'c', 9]
>>> print(mylst[:7])
[1, 12, 3.3, 8, 'a', 67, 33]
>>> print(mylst[::2])
[1, 3.3, 'a', 33, 9]
>>> print(mylst[::-1])
[9, 'c', 33, 67, 'a', 8, 3.3, 12, 1]

>>> print(mylst[9])  #out of range index
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> print(mylst[7:12])  #out of range
slice
['c', 9]
```



- Slice always returns a sequence
  Difference between `mylst[0]` and `mylst[0:1]`

- Difference between `tuple` and `list`
  `list` elements can be added, changed or removed after creation
  `tuple` cannot be changed once it is created
  Neither can `str`

```
>>> mytuple[0] = 'new first element'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>> mylist[0] = 'new first element'
>>> print(mylist)
['new first element', 12.22, b'\xaf']
```

```
>>> mystr = 'abcd'
>>> mystr[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Python

Sequence Types

Some operations on sequence types

```
>>> a = 'hello' # concatenation
>>> a = a + 'world'
>>> print(a)
helloworld

>>> mylst = ['a','b',123]
>>> mylst = mylst + ['newlast']
>>> print(mylst)
['a', 'b', 123, 'newlast']
```

```
>>> a = "hello"*4 #repetition
>>> print(a)
hellohellohellohello

>>> a = (1,2,3)*4
>>> print(a)
(1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3)
```

Python

Sequence Types

Membership testing

- Using the `in` keyword

```
>>> mylst = ['word', 12.2, 33]
>>> print('wo' in mylst)
False
>>> print('word' in mylst)
True
>>> print(33 in mylst)
True
>>> mystr = 'multiple words'
>>> print('lti' in mystr)
True
```

Iterating over sequence types

- A sequence type is an ordered set of values/objects
- An iterator returns each value in a sequence one after the other
- Iterators can be obtained from them using the `iter()` built-in function which returns an iterator
- Each value can be returned from an iterator by calling the `next()` built-in function with an iterator argument

Python

Iterators

```
>>> mytuple = (1.22, 2+3j, ['a','b'])
>>> tup_it = iter(mytuple)
>>> type(tup_it)
<class 'tuple_iterator'>

>>> print(next(tup_it))
1.22
>>> print(next(tup_it))
(2+3j)
>>> print(next(tup_it))
['a', 'b']

>>> print(next(tup_it))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Iterating over sequence types

- Iterators cannot be “reset”
  New iterator needs to be created in order to obtain values from the beginning
- Iterators raise the `StopIteration` exception to signal to the caller that there are no more values to return

Python

loops

`for` statement

- Executes a suite of statements for each item in an iterator after assigning the item to a loop variable

```
for <var> in <seq-type/iterator>:
    statement-1
    statement-2
    Statement-3
```

- Statements 1-3 are executed for each value of <var> from the sequence or iterator

Python

loops

for statement

```
>>> mynums = (0,1,2,3,4,5,6)
>>> for num in mynums:
...     print('square of', num, 'is', num**2)
...
square of 0 is 0
square of 1 is 1
square of 2 is 4
square of 3 is 9
square of 4 is 16
square of 5 is 25
square of 6 is 36
```

`for` statement

`range()` built-in function can be used to write loops like the one above more concisely

```
>>> for num in range(7):
...     print('square of', num, 'is', num**2)
...
square of 0 is 0
square of 1 is 1
square of 2 is 4
square of 3 is 9
square of 4 is 16
square of 5 is 25
square of 6 is 36
```

See `help(range)`

Algorithm: find maximum of n numbers and its position

Let list of numbers be numlist and let x represent the largest integer and p its position

1. Let x = numlist[0] and p = 0
2. Repeat for k = 0 to k = n-1
   If numlist[k] > x then x = numlist[k] and p = k

- `len()` is a built-in function that returns the number of items in a sequence object

```
maxnum = numlist[0]
maxpos = 0
for idx in range(len(numlist)):
    if numlist[idx] > maxnum:
        maxnum = numlist[idx]
        maxpos = idx
print(maxnum,"at",maxpos)
```

- `enumerate()` built-in function allows us to implement it in a more *pythonic* way

```
maxnum = numlist[0]
maxpos = 0
for idx, num in enumerate(numlist):
    if num > maxnum:
        maxnum = num
        maxpos = idx
print(maxnum,"at",maxpos)
```

`while` statement

- Executes a suite of statements as long as a condition is true

```
while <expression>:
    statement-1
    statement-2
    statement-3
```

- Statements 1-3 are executed repeatedly as long as `<expression>` evaluates to True

Fibonacci Sequence

Is a sequence of integers with the characteristic that each integer is the sum of the two preceding it

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
```

Loops

Fibonacci Sequence

- A program to generate the fibonacci numbers less than a value

```
1    n = input('enter number:')
2    n = int(n)

3    a, b = 0, 1
4    while b < n:
5        print(b)
6        a, b = b, a+b
```

- `input()` reads a line from stdin and returns a str
- `int()` returns an `int` given a `int`, `float` or `str` argument
- Tuple as target of assignment called *tuple unpacking*

Loops

Fibonacci Sequence

```
$ python3 fibonacci.py
enter number:120
1
1
2
3
5
8
13
21
34
55
89
```

`break` and `continue`

`break` will terminate the loop within which it is enclosed and `continue` will cause the loop to skip to the next iteration

Loops

Break and continue

```
1    while True:
2        inp = input('Enter a word: ')
3        if inp == 'end' or inp == 'End':
4            break
5        elif inp == 'skip' or inp == 'Skip':
6            continue
7        else:
8            print(inp)
```

```
$ python3 break_continue.py
Enter a word: test
test
Enter a word: test with words
test with words
Enter a word: skip
Enter a word: Skip
Enter a word: one more
one more
Enter a word: end
$
```

Python

Review

- Sequence Types
  `str, bytes, tuple, list`
- Indexing
  With integers or slices
- Common operations
  Concatenation (+), repetition (*), membership testing, `len()` built-in, assigning to a list index
- Iterators from sequences
- `for` statement and `range()` and `enumerate()` built-in functions

Methods of Sequence Types

- Many operations on sequence types are defined as *methods*
- Methods are functions that are *attributes* to a variable

  i.e They are accessed with the dot . operator

  `<object/variable>.method(<args))`

Methods of `str` Type

Close to 50 utility methods defined. We examine a few

- str methods return a new string

  Strings cannot be changed once created. Hence cannot be modified in place

Methods

str

- `upper()`
  Return string with characters in uppercase

```
>>> a = 'this is a sentence'

>>> print(a.upper())
THIS IS A SENTENCE

>>> print(a)
this is a sentence

>>> a = a.upper()
>>> print(a)
THIS IS A SENTENCE
```

Methods

str

- `find()`
  Locate a substring

```
>>> a = 'this is a sentence'

>>> print(a.find('is'))
2
```

Methods

str

- `isalpha(), isdigit()`
  Test whether string has only letter or only digits

```
>>> print('wordwithnospaces'.isalpha())
True

>>> print('wordwithnospaces'.isdigit())
False

>>> arnum = '٥٠٦١'
>>> print(arnum.isdigit())
True
```

`dir()` and `help()`

- `dir()` lists all attributes bound to a variable or object
- `help()` show detailed documentation
  See `help(str)` for a list of all methods and their description

Methods of `list` Type

- `list` methods that modify the list operate in-place
  Do not return any result

Methods

list

- `append()`
  Add an element to the end of the list

```
>>> mylst = ['3', 3.0, 'string']

>>> mylst.append(['a','b'])
>>> print(mylst)
['3', 3.0, 'string', ['a', 'b']]
```

Methods

list

- `insert()`
  Insert object before index

```
>>> mylst = ['3', 3.0, 'string']

>>> mylst.insert(2, ['a','b'])
>>> print(mylst)
['3', 3.0, ['a', 'b'], 'string']
```

Methods

list

- `reverse()`
  Reverse the list in-place

```
>>> mylst = ['3', 3.0, 'string']

>>> mylst.reverse()
>>> print(mylst)
['string', 3.0, '3']
```

Methods

list

- `sort()`
  Sort the list in-place

```
>>> mylst = ['money', 'apple', 'cat', 'yes']

>>> mylst.sort()
>>> print(mylst)
['apple', 'cat', 'money', 'yes']
```

Methods

list

- `index()`
  Return index of first occurrence of value in sequence

```
>>> mylst = ['money', 'apple', 'cat', 'yes', 'cat']

>>> print(mylst.index('cat'))
2
```

- Implement using built-ins

Example

list

List of squares

Program to generate a list containing squares of `n` integers starting from `0`

```
n = 12

squares = []
for i in range(n):
    squares.append(i**2)
print(squares)
```

```
$ python3 squares.py
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

Example

List comprehension

- The previous construct is common

  So Python provides a shorter syntactic equivalent

```
>>> n = 12
>>> squares = [i**2 for i in range(n)]
>>> print(squares)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

```
$ python3 squares.py
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

- Inside square brackets

  An expression followed by a for clause followed by zero or more for or if clauses