Lecture 4: Data Types in Python

BT 3051 – Data Structures and Algorithms for Biology

Karthik Raman
Department of Biotechnology
Indian Institute of Technology Madras

August 11, 2014

Python Lists

Lists store a sequence of data, which supports **indexing**

```
>>> primes = [2, 3, 5, 7, 11]
>>> primes[0] #indices start at 0
2
>>> primes[4]
11
>>> primes[-1] #negative indexing!
11
>>> primes[:3]
[2, 3, 5]
>>> primes[:-1]
[2, 3, 5, 7]
>>> primes[5]
Traceback (most recent call last):
  File "<pyshell#47>", line 1, in <module>
    primes[5]
IndexError: list index out of range
>>> primes.append(13)
>>> primes
[2, 3, 5, 7, 11, 13]
```

Python Lists

```
>>> primes = [2, 3, 5, 7, 11]
>>> len(primes)
5
>>> max(primes)
11
>>> min(primes)
2
>>> for p in primes:
    print (p)
2
3
5
7
11
>>> if 3 in primes:
    print ('Prime')
Prime
>>> 3 in primes
True
```

Enumerating a List

```
primes = [2, 3, 5, 7, 11]

for i, val in enumerate(primes):
    print i, val
```

Cleaner and more concise than:

```
i = 0
for val in primes:
    print i, val
    i = i + 1
```

List comprehensions I

```
>>> range(5)
range(0, 5)
>>> list(range(5))
[0, 1, 2, 3, 4]
```

Make new lists by manipulating old ones:

```
>>> squares = [v*v for v in list(range(5))]
>>> squares
[0, 1, 4, 9, 16]
>>> squares = [v*v for v in range(5)]
>>> squares
[0, 1, 4, 9, 16]
```

List comprehensions II

One can also use an `if` statement:

```
>>> even_primes = [v for v in primes if v%2==0]
>>> even_primes
[2]
>>>
```

Nested comprehensions are also possible (expressive!):

```
>>> set_x = list(range(5))
>>> set_y = list(range(2))
>>> set_x_cross_y = [[x,y] for x in set_x for y in set_y
                     ]
>>> print(set_x_cross_y)
[[0, 0], [0, 1], [1, 0], [1, 1], [2, 0], [2, 1], [3, 0],
                                  [3, 1], [4, 0], [4, 1]]
```

Python Tuples

A tuple consists of a number of values separated by commas, for instance:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

The statement `t = 12345, 54321, 'hello!'` is an example of tuple packing: the values `12345, 54321` and `'hello!'` are packed together in a tuple. The reverse operation is also possible, e.g.:

```
>>> x, y, z = t
```

Tuples

Tuples are similar to lists and strings, but they are immutable — i.e. they cannot be modified

```
>>> tu1 = (0,0,0)
>>> tu2 = (1,1,1)
>>> tu1[1] = 1
Traceback (most recent call last):
  File "<pyshell#118>", line 1, in <module>
    tu1[1] = 1
TypeError: 'tuple' object does not support item
                                    assignment
>>> print(tu1[0])
0
>>> x1,y1,z1 = tu1
>>> x1
0
>>> y1
0
>>> z1
0
```

Strings

- Strings are similar to lists — can be indexed, sliced
- A horde of built-in commands (recall `dir (str)`)

```
>>> str = 'Hello, world!'
>>> str[0]
'H'
>>> str[-1]
'!'
>>> str[:5]
'Hello'
>>> str[:5]+str[5:]
'Hello, world!'
>>> str+='!!'
>>> str
'Hello, world!!!'
>>> str[2]='L'
Traceback (most recent call last):
  File "<pyshell#132>", line 1, in <module>
    str[2]='L'
TypeError: 'str' object does not support item assignment
```

Strings

Parsing strings

```
>>> data = '(1, 2, 3, 4, 5, 6)'
>>> data = data.lstrip('(')
>>> data = data.rstrip(')')
>>> nums = data.split(',')
>>> nums
['1', ' 2', ' 3', ' 4', ' 5', ' 6']
>>> data = [int(n) for n in nums]
>>> data
[1, 2, 3, 4, 5, 6]
```

In one line,

```
>>> print([int(n) for n in data.strip('()').split(',')])
[1, 2, 3, 4, 5, 6]
```

More string functions

```
>>> str = 'Hello, world!'
>>> len(str)
13
>>> str.upper()
'HELLO, WORLD!'
>>> str.title()
'Hello, World!'
>>> ' '.join(['Hello','World!','Bye.'])
'Hello World! Bye.'
```

Dictionaries

- Very powerful data structure!
- Also known as "associative arrays"
- Maps from a set of keys to a set of values
- $K = \{ \text{ keys } \}$, $V = \{ \text{ values } \}$. $D : K \to V$:

$$k \xmapsto{D} v_k \in V$$

- In Python, you create D by a series of insertions of tuples $(k, v_k) \in K \times V$.
- It is "fast" to compute $D(k)$
- Dictionaries are particularly useful in biology!

Dictionaries

- Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any non-mutable type; strings and numbers can always be keys
- Tuples can be used as keys if they contain only strings, numbers, or tuples
- Can't use lists as keys, since lists can be modified in place using `append()`
- Dictionary is an unordered set of key:value pairs — keys must be unique

Dictionaries

A pair of braces creates an empty dictionary:

```
amino = {}
```

A comma-separated list of `key : value` pairs within the braces adds initial `key : value` pairs to the dictionary:

```
amino = {'A' : 'Ala', 'R': 'Arg', 'K': 'Lys', 'F':'Phe'}
```

This is also the way dictionaries are written on output:

```
{'R': 'Arg', 'F': 'Phe', 'A': 'Ala', 'K': 'Lys'}
```

Note that the keys are stored in arbitrary order!

Dictionaries

Main operations

```
>>> amino['R']
'Arg'
```

It is an error to extract a value using a non-existent key:

```
>>> amino['S']
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    amino['S']
KeyError: 'S'
```

More keys can be added:

```
>>> amino['W'] = 'TRP'
>>> amino
{'W': 'TRP', 'R': 'Arg', 'F': 'Phe', 'A': 'Ala', 'K': '
                                    Lys'}
```

Dictionaries

Changing and retreiving

```
>>> amino['W'] = 'Trp' #Over-writing an existing key
>>> del amino['R']
>>> amino
{'W': 'Trp', 'F': 'Phe', 'A': 'Ala', 'K': 'Lys'}
>>> amino.keys()
dict_keys(['W', 'F', 'A', 'K'])
```

Python 3.x has thrown away has_key:

```
>>> amino.has_key('G')
Traceback (most recent call last):
  File "<pyshell#48>", line 1, in <module>
    amino.has_key('G')
AttributeError: 'dict' object has no attribute 'has_key'
>>> 'R' in amino # we deleted it!
False
>>> 'W' in amino
True
```

Translating Text

```
>>> complement = {'A':'T', 'T':'A', 'C':'G', 'G':'C'}
>>> DNA = 'ATTAGCGCTTA'
>>> cDNA = [complement[base] for base in DNA]
>>> cDNA
['T', 'A', 'A', 'T', 'C', 'G', 'C', 'G', 'A', 'A', 'T']
>>> cDNA = ''.join([complement[base] for base in DNA])
>>> cDNA
'TAATCGCGAAT'
```

How would you do it without associative arrays?

Self-assessment Exercise: Can you find the three most frequent words in some WikiPedia article?

importing modules

```
>>> import math
>>> dir(math)
```

```
['__doc__', '__loader__', '__name__', '__package__',
 '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan'
, 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh'
, 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', '
fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum',
 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan', '
ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', '
modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt',
 'tan', 'tanh', 'trunc']
```

```
>>> math.pi
3.141592653589793
```

importing modules

```
>>> from math import pi
>>> pi
3.141592653589793
```

We can import a library with a different name using as:

```
>>> import math as m
>>> m.factorial(40)
815915283247897734345611269596115894272000000000
>>> m.pow(2,3)
8.0
>>> m.pow(pi,2)
9.86960401089358
```

We can also import everything from `math` into the current *namespace*:

```
>>> from math import *
>>> cos(pi)
-1.0
```

__main__ in Python

Why do Python programs have snippets like:

```
if __name__ == '__main__':
    main()
```

Read more about importing modules ...

Remember ...

- Practice, practice, practice!
- Document your files/functions
- Comment your code
- Write test cases (make it a habit!)
- How to write good test cases?
- Learn python idioms

Self-assessment Exercise

- Given a stretch of DNA (5'->3'), predict what will be the outcome (both strands), on digestion with the restriction endonuclease *EcoRI*
- Outcome:
  - Practice string manipulations in Python
  - Actual biological application
  - Write functions and return values
  - Implement `doctest` for some cases