University of Zurich[UZH]

URPP Evolution in Action

URPP tutorial

Python part – basics part 2

Dr. Heidi E.L. Lischer
University of Zurich
Switzerland

17 April, 2015

Tutorial overview

- Object oriented programming
- Biopython
  - is an international association of developers of freely available Python tools for computational molecular biology

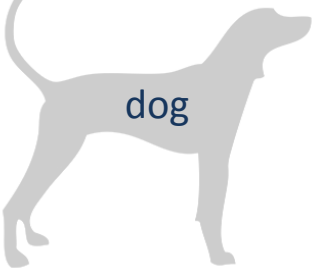
Object oriented programming

- Biopython is object-oriented
  - → Some knowledge about object oriented programming helps to understand biopython
- Procedural programming
  - What we did until now
    - → Set of instructions you follow from start to finish in order to complete a task
  - If you start having complicated nested variables (like arrays of hashes,…)
    → use object oriented programming
- Object oriented programming (OOP)
  - Way to organize data
  - Methods that work on the data
  - Put objects at the center of the process
  - → Helps to structure and organize the code

Classes

- Object oriented programs constructs objects according to the class definitions of the program
- Class:
  - Is a frame/template for creating objects
  - Specifies how an object can contain and process data
  - Specify two things:
    - Attributes: data holders → variables that contain data
    - Methods: functions for this class

Class:

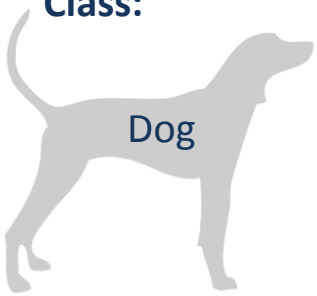


Attributes:	**Methods:**
Colour	Sit
Height	Bring
Length	Grow
Weight	

Objects

- Object:
  - Is an instance of a class → inherits the properties of the class
  - Act individually to the other objects of that class

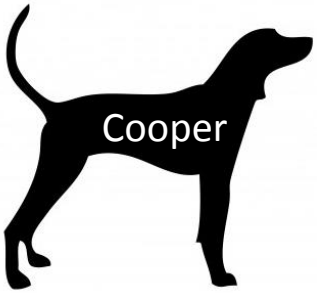
Class:



Attributes:
Colour
Height
Length
Weight

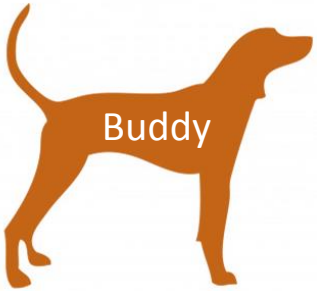
Methods:
Sit
Bring
Grow

Objects:



Attributes:
Colour: black
Height: 45 cm
Length: 90 cm
Weight: 15 kg

Methods:
Sit
Bring
Grow



Attributes:
Colour: brown
Height: 30 cm
Length: 55 cm
Weight: 10 kg

Methods:
Sit
Bring
Grow

Example

```
class Dog:
  def __init__(self, colour, height, length, weight):
    self.Colour = colour
    self.Height = height
    self.Length = length
    self.Weight = weight

  def sit(self):
    print 'Dog sit down'

  def bring(self, thing):
    print 'Dog bring the ', thing

  def grow(self, increaseHeight, increaseLength):
    self.Height += increaseHeight
    self.Length += increaseLength
```

Create class

initialize the instance variables of the object

Define methods
→ first parameter **has to be** "self"

→ DogClass.py

```
>>> import DogClass
>>> cooper = DogClass.Dog("black", 45, 90, 15)
>>> cooper.grow(3, 5)
>>> cooper.Height
48
```

Create object

Call method (without "self")

Encapsulation of Data

- We can directly access the attributes from outside, If the identifier doesn't start with two underscore character "__"
- Encapsulation:
  - mechanism for restricting the access to some of an object's components
  - internal representation of an object can't be seen from outside of the objects definition

```
class Dog:
  def __init__(self, colour, height, length, weight):
    self.Colour = colour
    self.Height = height
    self.Length = length
    self.__Weight = weight
...
```

Private, can't be seen or accessed from outside

Inheritance

- Classes can inherit attributes and methods from other classes
  → hierarchical relationship between classes

Super-class:

Attributes:
Colour
Height
Length
Weight

Methods:
Sit
Bring
Grow

Sub-class:

Attributes:
Speed

Methods:
Race

Sub-class:

Attributes:
Police number

Methods:
Search drugs
Hunt thieves

Inheritance

```
class Dog:
  def __init__(self, colour, height, length, weight):
    self.Colour = colour
    self.Height = height
    self.Length = length
    self.Weight = weight

  def sit(self):
    print 'Dog sit down‘
 ...

class RacingDog(Dog):
  def __init__(self, colour, height, length, weight, speed)
    Dog.__init__(self, colour, height, length, weight)
    self.Speed = speed

  def race(self, startTime):
    print 'Race will start at ', startTime
```

Super-class

Super-class in parenthesis

Sub-class

Biopython

- Goal of Biopython:
  - make it as easy as possible to use Python for bioinformatics by creating high-quality, reusable modules and classes

- Includes:
  - A standard sequence class → work with sequences
  - Sequence annotation
  - Population genetics
  - Phylogenetics
  - Parsers for various Bioinformatics file formats (FASTA, Genbank, BLAST, ...)
  - Interfaces to programs (BLAST, Clustalw...)
    - Local and remote BLAST
  - Access to online databases
    - Entrez
    - SwissProt
  - Genome graphics
  - many more

Heidi

Stefan

Installing Biopython

- Download Biopython: http://biopython.org/wiki/Download
- Linux:
  - Install from source:

```
python setup.py build
python setup.py test
sudo python setup.py install
```

  - Ubuntu package manager:

```
sudo apt-get install python-biopython
```

- Check version of Biopython:

```
import Bio
print Bio.__version__
```

Load Biopython

Sequences

- Central object in bioinformatics is the sequence → Seq class
- Seq object is not like a simple Python string
  - Different methods (e.g. translate(), complement(),…)
  - Has an alphabet attribute: object which defines the meaning
    - → Defined in Bio.Alphabet module

alphabet	type	description
IUPACProtein	Protein	basic
ExtendedIUPACProtein		- Additional elements (U, O) - ambiguous symbols
IUPACUnambiguousDNA	DNA	Basic DNA letters
IUPACAmbiguousDNA		Ambiguous letters
ExtendedIUPACDNA		Letters for modified bases
UPACUnambiguousRNA	RNA	Basic RNA letters
UPACAmbiguousRNA		Ambiguous letters

Sequences
- Alphabets

- Example:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("AGTACACTGGT", IUPAC.unambiguous_dna)
>>> my_seq
Seq('AGTACACTGGT', IUPACUnambiguousDNA())
>>> print(my_seq)
AGTACACTGGT
>>> my_seq.alphabet
IUPACUnambiguousDNA()
```

Import Seq class

Import IUPAC alphabet class

Create Seq object with alphabet

- Many string functions also work for Seq objects

```
>>> print len(my_seq)
11
>>> print my_seq[2]     #print third letter
T
>>> my_seq.count('G')
3
```

Sequences
- subsequencs

- Get a subsequence:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("AGTACACTGGT", IUPAC.unambiguous_dna)
>>> sub_seq = my_seq[3:8]
>>> sub_seq
Seq('ACACT', IUPACUnambiguousDNA())
```

→ 0-based

→ Start included, end not

→ new object produced is another Seq object!
(same alphabet as original Seq object)

Sequences
- concatenation

- Seq objects can be concatenated (like strings)
  → if they have the same alphabets

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> seq1 = Seq("AGTACACTGGT", IUPAC.unambiguous_dna)
>>> seq2 = Seq("CGTAGAATT", IUPAC.unambiguous_dna)
>>> seq1 + seq2
Seq('AGTACACTGGTCGTAGAATT', IUPACUnambiguousDNA())
>>> protein_seq = Seq("EVRNAK", IUPAC.protein)
>>> seq1 + protein_seq
Traceback (most recent call last):
...
TypeError: Incompatible alphabets IUPACUnambiguousDNA()
and IUPACProtein()
```

Sequences
- change case

- Changing case (alphabet aware): `upper()` or `lower()`

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> seq1 = Seq("acgtACGT", IUPAC.unambiguous_dna)
>>> seq1.upper()
Seq('ACGTACGT', IUPACUnambiguousDNA())
>>> seq1.lower()
Seq('acgtacgt', DNAAlphabet())
```

→ Strictly, IUPAC alphabets are only for upper case sequences

→ Thus it is transformed to a generic DNA alphabet

Sequences
- reverse complements

- Nucleotide sequences can be (reverse) complemented:
  - `complement()`
  - `reverse_complement()`

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> seq1 = Seq("AGTACACTGGT", IUPAC.unambiguous_dna)
>>> seq1.complement()
Seq('TCATGTGACCA', IUPACUnambiguousDNA())
>>> seq1.reverse_complement()
Seq('ACCAGTGTACT', IUPACUnambiguousDNA())
```

  - Reverse:

```
>>> seq1[::-1]
Seq('TGGTCACATGA', IUPACUnambiguousDNA())
```

→ `str[start:end:stepSize]`
→ stepSize = -1: reverse

Sequences
- transcription

- Nucleotide sequences can also be transcribed (DNA → mRNA) :
  - `transcribe()` → switch T to U
    → adjust alphabet

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> dna = Seq("ATGGCCGGCCGCTGAAAGGGTGCCCGATAG", IUPAC.unambiguous_dna)
>>> mRNA = dna.transcribe()
>>> mRNA
Seq('AUGGCCGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())
```

  - `back_transcribe()` → mRNA to DNA

Sequences
- translation

- Translation (mRNA → protein sequence): `translate()`

```
>>> protein = mRNA.translate()
>>> protein
Seq('MAGR*KGAR*', HasStopCodon(IUPACProtein(), '*'))
```

  – Internal stop codon → maybe wrong translation table used

- Use different translation table: `translate(table=codeTable)`
  – Based on NCBI codon tables

```
>>> mRNA.translate(table="Vertebrate Mitochondrial")
Seq('MAGRWKGAR*', HasStopCodon(IUPACProtein(), '*'))
```

- Translate until stop codon: `translate(to_stop=True)`

```
>>> mRNA.translate(to_stop=True)
Seq('MAGR', IUPACProtein())
```

→ Stop codon is not translated

Sequences
- compare

- Comparison of sequences is not straight forward:
  - Meaning of letters are context dependent

    →DNA 'ACGT' is not equal to protein 'ACGT‘
  - Biopython < 1.65: Compare string and alphabets

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> seq1 = Seq("ACGT", IUPAC.unambiguous_dna)
>>> seq2 = Seq("ACGT", IUPAC.ambiguous_dna)
>>> seq1 == seq2
False
```

→doesn’t solve the problem, as we expect that this should be true

  - Way around: compare sequences as strings → responsibility given to user

```
>>> str(seq1) == str(seq2)
True
```

Sequences
- compare

- Biopython 1.65: only compares sequence

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> seq1 = Seq("ACGT", IUPAC.unambiguous_dna)
>>> seq2 = Seq("ACGT", IUPAC.ambiguous_dna)
>>> seq1 == seq2
True
```

→ gives a warning if you compare incompatible alphabets (e.g. DNA vs. RNA)

```
>>> from Bio.Alphabet import generic_protein
>>> prot_seq = Seq('ACGT', generic_protein)
>>> seq1 == prot_seq
BiopythonWarning: Incompatible alphabets DNAAlphabet() and
ProteinAlphabet()
True
```

SeqRecord class

- Sequences sometimes have other higher level features associated:
  - Identifiers
  - Common names
  - Descriptions → SeqRecord class
  - Annotations
  - Other Features

- SeqRecord class:
  - Usually not created by hand
    → FASTA or GenBank files are directly read into this format by the SeqIO class
  - Attributes:
    - seq
    - id
    - name
    - description
    - annotations → dictionary
    - letter_annotations → dictionary
    - dbxrefs → list
    - features → list

SeqRecord
- create

- Create a SeqRecord:
  - You need a Seq object at minimum
  - add id, name and description → if not they will be assigned as unknown
  - and other features like annotations

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> seq = Seq("AGTACACTGGT", IUPAC.unambiguous_dna)

>>> from Bio.SeqRecord import SeqRecord
>>> seqRec = SeqRecord(seq)
>>> seqRec.id
'<unknown id>'
>>> seqRec.id = "AC12345"
>>> seqRec.description = "My test sequence"
>>> print(seqRec.description)
My test sequence
>>> seqRec.seq
Seq('AGTACACTGGT', IUPACUnambiguousDNA())
```

SeqRecord
- SeqFeature object

- SeqFeature object:
  - Organize and easily get feature information
  - Heavily based on the GenBank/EMBL feature tables
  - Key idea: describe a region on a sequence

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> from Bio.SeqFeature import SeqFeature, FeatureLocation
>>> seq = Seq("ACCGAGACGGCAAAGGCTAGCATAGGTATGAG", IUPAC.unambiguous_dna)
>>> feature1 = SeqFeature(FeatureLocation(5, 18), type="gene", strand=-1)

>>> from Bio.SeqRecord import SeqRecord
>>> seqRec = SeqRecord(seq, features=[feature1])

#extract coding region and reverse complement it (strand: -1)
>>> feature_seq = seqRec.features[0].extract(seqRec)
>>> print(feature_seq)
AGCCTTTGCCGTC
```

extract method takes care of everything

SeqRecord
- format method and subrecord

- format() method of SeqRecord:
  - Returns a formatted string containing your record (formats supported by Bio.SeqIO)

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Alphabet import generic_protein

>>> record = SeqRecord(Seq("MMYQQGCFAGGTVLRLAKDLAENNRGA", generic_protein),
                       id="gi|14150838|gb|AAK54648.1|AF376133_1",
                       description="chalcone synthase [Cucumis sativus]")
>>> print(record.format("fasta"))
>gi|14150838|gb|AAK54648.1|AF376133_1 chalcone synthase [Cucumis sativus]
MMYQQGCFAGGTVLRLAKDLAENNRGA
```

- You can extract parts of records
  - Any per-letter annotation is also sliced
  - Any feature that fall completely within the new sequence is preserved (and location is adjusted)

```
>>> subRecord = record[10:20]
```

Population genetics

- Biopython provides a module to do population genetics: `Bio.PopGen`
  - GenePop file format parser/writer
  - Methods to modify GenePop records
  - Coalescent simulations by Fastsimcoal2 (backward model of population genetics)
  - Methods to run FDist (Detecting selection and molecular adaptation)

→ More details under http://biopython.org/wiki/PopGen

Phylogenetics

- Module work with phylogenetics trees: `Bio.Phylo`
  - Tree object:
    contains global information about the tree (e.g.: rooted/unrooted)

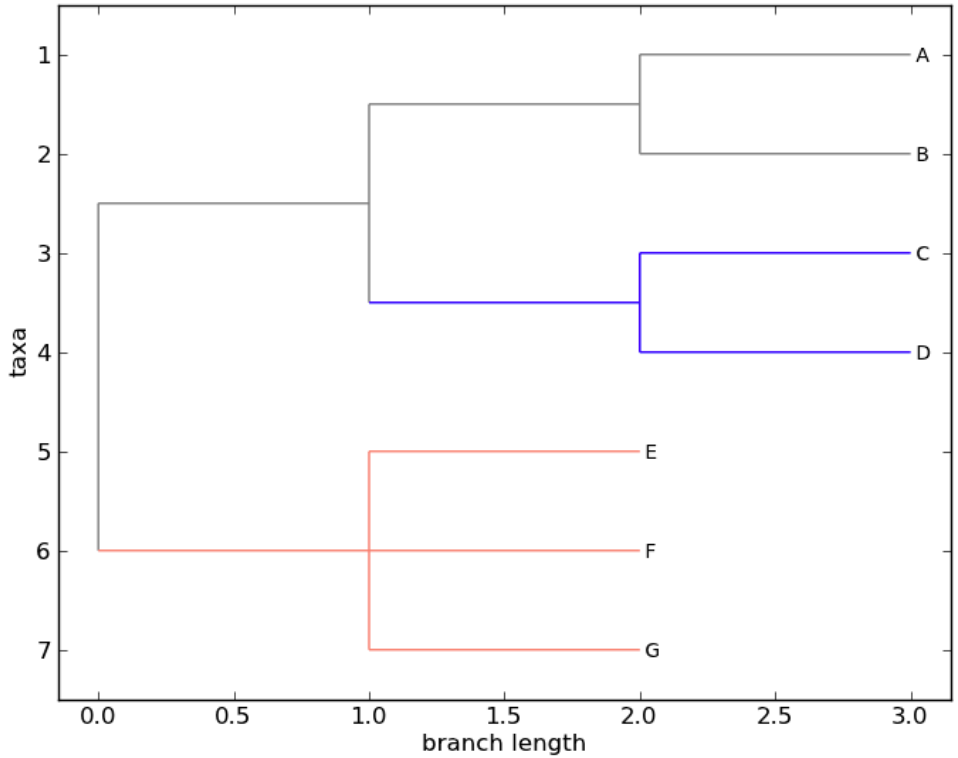
```
(((A,B),(C,D)),(E,F,G));
```

→ tree.dnd

```
>>> from Bio import Phylo
>>> tree = Phylo.read("simple.dnd", "newick")
>>> Phylo.draw_ascii(tree)
                                  ________________ A
                   ________________|
                  |                |________________ B
   ________________|
  |                |                ________________ C
  |                |________________|
_ |                                 |________________ D
  |
  |                ________________ E
  |                |
  |________________|________________ F
                   |
                   |________________ G
```

Phylogenetics

- Methods draw and drew_graphviz support many more display options:
  - Different branch colors
  - Branch widths



Phylogenetics

- Functions included:
  - Get parent of a clade
  - Index clades by name
  - Calculate distance between neighboring terminals
  - Convert between formats

→ More details under http://biopython.org/wiki/Phylo_cookbook

Acknowledgment

- Sources:
  - http://www.python-course.eu/object_oriented_programming.php
  - http://code.tutsplus.com/articles/python-from-scratch-object-oriented-programming--net-21476

- Biopython:
  - http://biopython.org/DIST/docs/tutorial/Tutorial.html