Python Programming:

- Interactive and batch modes
- Basic data types

Interactive and Batch modes

There are two ways to use Python: **interactive** and **batch** mode.
Both methods are complementary and they are used with different purposes:

•Interactive mode allows the programmer to get an immediate reply to each instruction.
•In batch mode, instructions are stored in one or more files and then executed. This is the standard way of running Python programs.

Interactive mode is used mostly for **small tests** while most programs are run in batch mode.

Interactive mode

In the interactive mode we enter one command or instruction at the time, after the command prompt “>>>”. Python will execute our command when we press the ENTER key:

```
>>> print "Hello World!"
Hello World!
```

Basic Input and Output

1) Output with **print**:

```
>>> print "Hello World!"
Hello World!
```

2) Input with **raw_input()** function:

```
>>> name = raw_input("Enter your name: ")
Enter your name: eun-jin
>>> print name
eun-jin
```

Note: a function is a module of code that takes some input data and generates a result. For instance, raw_input() takes a string to show to the user and returns the response from the user. Python has many functions we can use, and we can also create our own functions.

Interactive mode as a calculator

Interactive mode can be used as a calculator:
>>> 1+1
2
When ‘+’ is used on strings, it returns a concatenation:
>>> ’1’+’1’
’11’
>>> "A string of " + ’characters’
’A string of characters’

Note that single (’) and double (”) quotes can be used in an indistinct way, as long as they are used with consistency. That is, if a string definition is started with one type of quotes, it must be finished with the same kind of quote.

Different data types can’t be added:

```
>>> ’The answer is ’ + 42
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: cannot concatenate ’str’ and ’int’ objects
```

Only elements of the same type can be added. To convert this into a sum of strings, the number must be converted into a string, this is done with the str() function:

```
>>> ’The answer is ’ + str(42)
’The answer is 42’
```

Mathematical Operations

Any standard mathematical operation can be done in the Python shell:

```
>>> 12*2
24
>>> 30/3
10
>>> 2**8/2+100
228
```

Arithmetic-Style Operators

Symbol	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation
%	Modulus (remainder)

Precedence order

Operator precedence is the same as used in math. An easy way to remember precedence order is with the acronym PEMDAS:

P Parentheses have the highest precedence and are used to set the order of expression evaluation. This is why 2 * (3-2) yields 2 and (3-1) ** (4-1) yields 8. Parentheses can also be used to make expressions easier to read.
E Exponentiation is the second in order, so 2**2+1 is 5 and not 8.
MD Multiplication and Division share the same precedence. 2*2-1 yields 3 instead of 2.
AS Addition and Subtraction also share the same (latest) order of precedence.

Last but not least, operators with the same precedence are evaluated **from left to right**. So 60/6*10 yields 100 and not 1.

Note about Division

This may not be expected:

```
>>> 10/3
3
```

Division returns the floor, that is, the integer part of the result. To get the floating point result, at least one operand must be float:

```
>>> 10.0/3
3.3333333333333335
>>> 10/3.
3.3333333333333335
```

Batch mode

Most programs are stored in files. The code used in an interactive session can be accessed only when the session is active. Each time that an interactive session is closed, all typed code is gone. In order to have code persistence, programs are stored in text files. When a program is executed from such a text file, rather than line by line in an interactive interpreter, it is called batch mode.

These are regular text files usually with the “.py” extension. These files can be generated with any standard text editor (as Windows Notepad). We will use the editor included with Python, IDLE.

simple.py:

```
seq = raw_input('Enter DNA sequence: ')
print "The length of the sequence is: ", len(seq)
```

Comments

The “#” character has special significance for Python. It is used to identify lines that aren’t executed by the interpreter.

As a result, the lines that begin with that symbol are called “comments.” Comments don’t add functionality to the program but help the programmer or other possible readers of the code.

Let’s look at the previous program with a comment:

```
# We ask a sequence from the user
seq = raw_input('Enter DNA sequence: ')
# Now we print the length of the sequence
print "The length of the sequence is: ", len(seq)
```

Indentation

One of the first things that stands out to programmers about Python is its code indentation system. Non-programmers must be wondering at this point what is indentation of the source code. Here is some C code that is not indented:

```
if (attr == -1){while (x < 5){
printf("Waiting...\n");wait(1);
x = x+1;}printf("Everything is OK\n");}
else {printf("There is an error\n");}
```

Same program portion (or “code snippet”), but indented:

```
if (attr == -1) {
    while (x < 5) {
        printf("Waiting...\n");
        wait(1);
        x = x + 1;
    }
    printf("Everything is OK\n");
}
else {
    printf("There is an error\n");
}
```

Which one seems to be more clear?

Even not knowing C we can say that the second program is “clearer” than
the first. The following code snippet in Python is equivalent to the previous snippet in C:

```
if attr == -1:
    while x < 5:
        print("Waiting...")
        wait(1)
        x = x + 1
    print("Everything is OK")
else
    print("There is an error")
```

Data Types in Python

Here we will see different basic data types we can use in Python. One such fundamental data structure is a **sequence**. In a sequence, the data elements have a sequential order. Examples of sequences:

- Strings
- Lists
- Tuples

There are also unordered data types:

- Dictionaries
- Sets

Strings

A string is a type of sequence of characters delimited by a single quote (’), double quotes (”), single triple quotes (”’), or double triple quotes (”””). Therefore, the following strings are equivalent:

"This is a string in Python"
’This is a string in Python’
”’This is a string in Python”’
"""This is a string in Python"""

We cannot mix quotes:

```
>>> "Mixing quotes leads to the dark side’
  File "<stdin>", line 1
    "Mixing quotes leads to the dark side’
                                        ^
SyntaxError: EOL while scanning single-quoted string
```

Note: EOL stands for end-of-line.

Multi-line strings

Regarding strings enclosed by triple quotes, we can use them to indicate multi-line strings (also known as block string):

```
"""Hi! I’m a
multiline
        string"""
```

The character ’\n’ represents an end-of-line (EOL) character. Therefore, the code above could be written in one line as:

```
"Hi! I’m a\nmultiline\n        string"
```

Unicode strings

There are two types of strings: byte strings and Unicode strings.

•**Byte strings** contain bytes. In practice, English characters are stored as byte strings.
•**Unicode strings** are intended for text. “International” characters are stored as Unicode strings.

’I am a byte string in Python 2.5!!!’

u’I am UNICODE: π’

u' 제주 '

Note the u at the beginning of the unicode strings.

Byte and unicode are interconvertible as long as you specify an encoding scheme:

```
>>> s = u'Andrés'
>>> s.encode('utf-8')
'Andr\xc3\xa9s'
>>> b=s.encode('utf-8')
>>> x=unicode(b,'utf-8')
>>> print x
Andrés
>>> x==s
True
```

String Manipulation

Strings are **immutable:** once a string is created, it **can't be modified**. If you need to change a string, what you can do is to make a derivated string. This is done using the string as a parameter in a function and then get the returned value.

```
>>> signal_peptide="MASKATLLLAFTLLFATCIA"
```

To get a lower case version of the string, use the method lower():

```
>>> signal_peptide.lower()
'maskatlllaftllfatcia'
```

In spite of having obtained the string lower case, the original string has not been modified:

```
>>> signal_peptide
'MASKATLLLAFTLLFATCIA'
```

If we want this new lower case string to have the same name as the previous one, all we need to do is rename it:

```
>>> signal_peptide=signal_peptide.lower()
>>> signal_peptide
'maskatlllaftllfatcia'
```

Methods Associated with Strings

replace(old,new[,count]): Allows us to replace a portion of a string (old) with another (new). If the optional argument count is used, only the first count occurrences of old will be replaced.

Example:

```
>>> DNAseq="TTGCTAG"
>>> mRNAseq=DNAseq.replace("T","U")
>>> mRNAseq
'UUGCUAG'
```

count(sub[, start[, end]]): Counts how many times the substring sub appears, between start and end position (if available).

Example: Calculate the CG content of a sequence:

```
>>> c=DNAseq.count("C")
>>> g=DNAseq.count("G")
>>> float(c+g)/len(DNAseq)*100
48.387096774193552
```

find(sub[,start[,end]]): Returns the position of the substring sub, between start and end position (if available). If the substring is not found in the string, this method returns the value -1.

Example:

```
>>> mRNAseq.find("AUG")
17
```

split([sep [,maxsplit]]): Separates the “words” of a string and returns them in a list.

If a separator (sep) is not specified, the default separator will be the white space:
>>> "This string has words separated by spaces".split()
[’This’, ’string’, ’has’, ’words’, ’separated’, ’by’, ’spaces’]

When white space is not the data separator, we have to specify a custom separator:
>>> "Alex Doe,5555-2333,nobody@example.com".split()
[’Alex’, ’Doe,5555-2333,nobody@example.com’]

In this case the separator is a comma (“,”), so we have to state it explicitly:
>>> "Alex Doe,5555-2333,nobody@example.com".split(",")
[’Alex Doe’, ’5555-2333’, ’nobody@example.com’]

The inverse function of split() is join():

join(seq): Joins the sequence using a string as a “glue character”:
>>> ’;’.join([’Alex Doe’, ’5555-2333’, ’nobody@example.com’])
’Alex Doe;5555-2333;nobody@example.com’

To join a sequence without any glue character, use empty quotes (””):
>>> ’’.join([’A’,’C’,’A’,’T’])
’ACAT’

Lists

Lists are one of the most versatile object types in Python. A list is an ordered collection of objects. It is represented by elements separated by commas and enclosed between square brackets.

The next code shows how to define and name a list:
>>> first_list=[1,2,3,4,5]
This is a list with five elements. In this case, all the elements are of the same type (integer).

A list can hold different kinds of elements:
>>> other_list=[1,"two",3,4,"last"]

A list can even contain another list:
>>> nested_list=[1,"two",first_list,4,"last"]
>>> nested_list
[1, ’two’, [1, 2, 3, 4, 5], 4, ’last’]

An empty list is defined as empty brackets:

```
>>> empty_list=[]
>>> empty_list
[]
```

An empty list doesn’t have any use of its own, but sometimes we may want to define an empty list to add elements at a later time.

List Initialization

If you know in advance that a list is going to have five elements, we can initialize it with a default value:

```
>>> codons = [None] * 5
>>> codons
[None, None, None, None, None]
```

This type of list initialization can be useful when working with big lists and the number of elements is known beforehand.

List Comprehension

There is another way to define a list. A list can be created from another list. As in mathematics where you can define a set by enumerating all its elements (enumeration) or by describing properties enjoyed exclusively by its members (comprehension), in Python a list can be created by both methods:

```
>>> A = [0,1,2,3,4,5]
```
A list defined by comprehension in Python,
```
>>> [3*x for x in A]
[0, 3, 6, 9, 12, 15]
```

Accessing List Elements

As one of the other sequence data types, you access list elements by an index starting at zero.

```
>>> first_list=[1,2,3,4,5]
>>> first_list[0]
1
>>> first_list[1]
2
```

Another way of obtaining lists is by turning a non list object into a list by using the built-in function list():

```
>>> aseq="atggctaggc"
>>> list(aseq)
['a', 't', 'g', 'g', 'c', 't', 'a', 'g', 'g', 'c']
```

The function str() converts the input parameter into a string. Can we use it to revert the effect of list()?

```
>>> str(['a', 't', 'g', 'g', 'c', 't', 'a', 'g', 'g', 'c'])
'['a', 't', 'g', 'g', 'c', 't', 'a', 'g', 'g', 'c']'
```

Clearly the result is not for what we were expecting. The str() function turned the list into a string, but not into the original string. Instead, the result was a literal representation of the list. To have the original string back, we have to join the elements of the list. This can be done with the join() function:

```
>>> "".join(['a', 't', 'g', 'g', 'c', 't', 'a', 'g', 'g', 'c'])
'atggctaggc'
```

Copying a List

Copying a List

```
>>> a=[1,2,3]
>>> b=a
>>> b.pop()
3
>>> a
[1, 2]
```

As seen, “=” doesn’t copy the values, but works as it copy a reference to the original object. There are two ways to make an independent copy of a list:

Using the copy module:

```
>>>   import copy
>>>   a=[1,2,3]
>>>   b=copy.copy(a)
```

Without the copy module:

```
>>>   a=[1,2,3]
>>>   b=a[:]
>>>   b.pop()
3
>>>   a
[1,   2, 3]
```

Modifying Lists

Unlike strings, lists can be modified by either adding, removing, or changing their elements:

1) Adding
There are three ways to add elements into a list: append, insert, and extend.

append(element): Adds an element at the end of the list.

```
>>> first_list.append(99)
>>> first_list
[1, 2, 3, 4, 5, 99]
```

insert(position,element): Inserts the element element at the position position.

```
>>> first_list.insert(2,50)
>>> first_list
[1, 2, 50, 3, 4, 5, 99]
```

extend(list): Extends a list by adding a list to the end of the original list.

```
>>> first_list.extend([6,7,8])
>>> first_list
[1, 2, 50, 3, 4, 5, 99, 6, 7, 8]
```

This is the same as using the + symbol:

```
>>> [1,2,3]+[4,5]
[1, 2, 3, 4, 5]
```

2) Removing
There is also three ways to remove elements from a list.

pop([index]): Removes the element in the index position and returns it to the point where it was called. Without parameters, it returns the last element.

```
>>>  first_list
[1,  2, 50, 3, 4, 5, 99, 6, 7, 8]
>>>  first_list.pop()
8
>>>  first_list.pop(2)
50
>>>  first_list
[1,  2, 3, 4, 5, 99, 6, 7]
```

remove(element): Removes the element specified in the parameter. In the case where there is more than one copy of the same object in the list, it removes the first one, counting from the left. Unlike pop(), this function does not return anything.

```
>>> first_list.remove(99)
>>> first_list
[1, 2, 3, 4, 5, 6, 7]
```

Another way of removing an element of a list is using the command **del**, for what:
del first_list[0]
Has a similar effect to:
first_list.pop(0)
With the difference that pop() returns the extracted element where it was called, while del just deletes it.

TABLE 3.1: Common List Operations

Properties	**Description**
s.append(x)	Adds the x element to list s
s.count(x)	Counts how many times x is in s
s.index(x)	Returns where is x in list s
s.remove(x)	Removes the element x from list s
s.reverse()	Reverse list s
s.sort()	Sort list s

Tuples

The main characteristic of the tuple is that once created, it cannot be modified. That is why they are referred to as “immutable lists.”

```
>>> point=(23,56,11)
```
point is a tuple with three elements (23, 56 and 11).

There is a particular case that you should use a trailing comma, when defining a tuple with one element:

```
lone_element_tuple = (5,)
```

Why the comma? Because otherwise the tuple is confused with the number 5.

You are not allowed to add or to remove elements from a tuple:

```
>>> point.append(3)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'

>>> point.pop()
Traceback (most recent call last):
File "<stdin>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'pop'
```

A typical example of a tuple is a coordinate system. In a three-dimensional coordinate system, each point is referred to by a three element tuple (x, y, z). The number of elements for each tuple does not changes.

Common Properties of the lists and tuples (sequences)

1) Indexing

Indexing was discussed when covering list, but for the sake of completeness, it is also here. Since the elements in the sequences are ordered, we can gain access to any element through an index that begins at zero:

```
>>> point=(23,56,11)
>>> point[0]
23
>>> point[1]
56
>>> sequence="MRVLLVALALLALAASATS"
>>> sequence[0]
'M'
>>> sequence[5]
'V'
>>> parameters=['UniGene','dna','Mm.248907',5]
>>> parameters[2]
'Mm.248907'
```

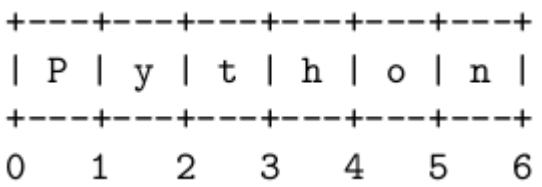
To access an element that is inside a sequence, which is itself inside another sequence, you need to use another index:

```
>>> seqdata=("MRVLLVALALLA",12,"5FE9EEE8EE2DC2C7")
>>> seqdata[0][5]
'V'
```

2) Slicing

You can select a portion of a sequence using slice notation. Slicing consists of using two indexes separated by a colon (:). These indexes represent a position in the existing space between the elements.

The string “Python” can be represented as,




```
>>> my_sequence="Python"
>>> my_sequence[0:2]
’Py’
```

When omitting the first sub index, the index value defaults to the first position (0):

```
>>> my_sequence[:2]
’Py’
```

On the other hand, when the second sub index is omitted, the index value defaults to the last position (-1):

```
>>> my_sequence="Python"
>>> my_sequence[4:6]
'on'
>>> my_sequence[4:]
'on'
```

There is a third, optional index to skip positions (step argument):

```
>>> my_sequence[1:5]
'ytho'
>>> my_sequence[1:5:2]
'yh'
```

step with a negative number is used to count backwards. So -1 (in the third position) can be used to invert a sequence:

```
>>> my_sequence[::-1]
'nohtyP'
```

Slicing always returns another sequence.

3) Membership Test

You can verify whether an element belongs to a sequence, using the in keyword:

```
>>> point=(23,56,11)
>>> 11 in point
True
>>> my_sequence="MRVLLVALALLALAASATS"
>>> "X" in my_sequence
False
```

4) Concatenation

You can concatenate two or more sequences of the same class using the “+” sign:

```
>>> point=(23,56,11)
>>> point2=(2,6,7)
>>> point+point2
(23,56,11,2,6,7)
>>> DNAseq="ATGCTAGACGTCCTCAGATAGCCG"
>>> TATAbox="TATAAA"
>>> TATAbox+DNASeq
’TATAAAATGCTAGACGTCCTCAGATAGCCG’
```

5) len, max, and min:

len() returns the amount of elements of a sequence:

```
>>> point=(23,56,11)
>>> len(point)
3
>>> my_sequence="MRVLLVALALLALAASATS"
>>> len(my_sequence)
19
```

The use of max() and min() is recommended to find the smallest and largest element in a sequence:

```
>>> point
(23, 56, 11)
>>> max(point)
56
>>> min(point)
11
```

max() and min() applied to strings returns a character according to the maximum or minimum value of its ASCII code:

```
>>> MySequence="MRVLLVALALLALAASATS"
>>> max(MySequence)
’V’
>>> min(MySequence)
’A’
```

Dictionaries

The main characteristic of a dictionary is that it stores arbitrary indexed unordered data types.

```
>>> IUPAC = {’A’:’Ala’,’C’:’Cys’,’E’:’Glu’}
>>> print "C stands for the amino acid ", IUPAC[’C’]
C stands for the amino acid Cys
```

IUPAC is the name of a dictionary with three elements. It was defined by enclosing is *key:value* pairs between curly brackets “{“ and “}”.

This dictionary works as a translation table that allows us to translate between the one-letter amino acid code to a three-letter code. Every element consists of a pair key:value. The key is the index used to retrieve the value:

```
>>> IUPAC['E']
'Glu'
```

Not every object can be used as a dictionary key, only immutable objects like strings, tuples and numbers can be used as keys. Lists cannot be used as a key, for example.

Creating dictionaries

1) The first method is by direct assignment:
IUPAC = {'A':'Ala','C':'Cys','E':'Glu'}

2) Using dict:
```
>>> species=[('human','homo sapiens'),('mouse','mus musculus'),('cow','bos taurus')]
>>> species_d = dict(species)
>>> species_d['mouse']
'mus musculus'
```

3) dict also accepts name=value pairs in the keyword argument list:
```
>>> species = dict('human'='homo sapiens','mouse'='mus musculus','cow'='bos taurus')
>>> species
{'human':'homo sapiens', 'mouse':'mus musculus', 'cow':'bos taurus'}
```

4) Another way to initialize a dictionary is to create an empty dictionary and add elements as needed:

```
>>> species = {}
>>> species['human'] = 'homo sapiens'
>>> species['mouse'] = 'mus musculus'
>>> species
{'human': 'homo sapiens', 'mouse': 'homo sapiens'}
```

To add values to a dictionary,

```
>>> IUPAC['S']='Ser'
>>> len(IUPAC)
4
```

Operating with Dictionaries

Dictionaries Are Made of Keys and Values. You can list the keys and values of dictionaries with the methods keys() and values() respectively:

```
>>> IUPAC.keys()
['A', 'C', 'E', 'S']
>>> IUPAC.values()
['Ala', 'Cys', 'Glu', 'Ser']
```

These methods can be used to check the presence of a key in a dictionary. keys() returns a list . You could use keys() as a possible check for the presence of an element in a list:

```
>>> 'Z' in IUPAC.keys() # Method not recommended!
False
>>> 'Z' in IUPAC
False
```

Another way of gaining access to the elements of a dictionary is by using items():

```
>>> IUPAC.items()
[('A', 'Ala'), ('C', 'Cys'), ('E', 'Glu'), ('S', 'Ser')]
```

items() returns a list with a tuple for every key/value pair.

Safe Query of Dictionary Values

A way to query a value from a dictionary, without the risk of invoking an exception, is to use get(k,x). K represents the key of the element to extract, while x is the element that will be returned in case k is not found as a key of the dictionary.

```
>>> IUPAC.get('A','No translation')
'Ala'
>>> IUPAC.get('Z','No translation')
'No translation'
```

If you omit x, and there is no k present in the dictionary, the method returns None.

```
>>> IUPAC.get('Z')
None
```

Erasing Elements

To erase elements from a dictionary, use the del instruction:

```
>>> del IUPAC['A']
>>> IUPAC
[('C', 'Cys'), ('E', 'Glu'), ('F', 'Phe')]
```

TABLE 3.2: Methods Associated with Dictionaries

Properties	**Description**
len(a)	Number of elements of a
a[k]	The element from a that has a k key
a[k] = v	Set a[k] to v
del a[k]	Remove a[k] from a
a.clear()	Remove all items from a
a.copy()	A copy of a
k in a	True if a has a key k, else False
k not in a	Equivalent to not k in a
a.has_key(k)	Equivalent to k in a, use that form in new code
a.items()	A copy of a's list of (key, value) pairs
a.keys()	A copy of a's list of keys
a.update([b])	Updates (and overwrites) key/value pairs from b
a.fromkeys(seq[, value])	Creates a new dictionary with keys from *seq* and values set to *value*
a.values()	A copy of a's list of values
a.get(k[, x])	a[k] if k in a, else x
a.setdefault(k[, x])	a[k] if k in a, else x (also setting it)
a.pop(k[, x])	a[k] if k in a, else x (and remove k)
a.popitem()	Remove and return an arbitrary (key, value) pair