Introductory Linux Course

Python I

Martin Dahlö – UPPMAX

Author: Nina Fischer
Dept. for Cell and Molecular Biology, Uppsala University

August, 2019
Outline

- Python basics – get started with Python
- Data types
- Control structures
  - Conditions: if-else
  - Loops: for and while
- Our first Python program
Python Basics

- We use Python 3 (version 3.6.0) in this course. For this we need to load a module:

  [dahlo@rackham1 ~]$ module load python3/3.6.0
Python Basics

● We use Python 3 (version 3.6.0) in this course. For this we need to load a module:

```
[dahlo@rackham1 ~]$ module load python3/3.6.0
```

● When you type `python3` on the command line the Python-Interpreter comes ready:

```
[dahlo@rackham1 ~]$ python3
Python 3.6.0 (default, Nov 21 2017, 09:52:46)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-18)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>}
```
Python Basics

• We use Python 3 (version 3.6.0) in this course. For this we need to load a module:

```bash
[dahlo@rackham1 ~]$ module load python3/3.6.0
```

• When you type `python3` on the command line the Python-Interpreter comes ready:

```bash
[dahlo@rackham1 ~]$ python3
Python 3.6.0 (default, Nov 21 2017, 09:52:46)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-18)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

**Note**: If you type "python", you start Python 2 (version 2.7.5)!
Python Basics

- With `quit()`, `exit()`, or `Ctrl-D` you can close the Python-interpreter

```
[dahlo@rackham1 ~]$ python3
Python 3.6.0 (default, Aug 4 2017, 00:39:18)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-16)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> quit()
[dahlo@rackham1 ~]$
```

- With arrow keys "up" (↑) and "down" (↓) you can scroll through previous commands
Python as Calculator

>>> 2 + 2
4
>>>
Python as Calculator

>>> 2 + 2
4

>>> 50 - 5*6
20

>>>
Python as Calculator

>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>>
Python as Calculator

```python
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 12.45 / 100 + 7.5e-3
0.132
```
• Output appears in the terminal (the window you are working in)
• With the `print()` function you can write text

```python
>>> print("Welcome to our Introductory Linux Course!")
Welcome to our Introductory Linux Course!
>>> print(2)
2
>>> print(2+2)
4
```
Python Assignment

• Store text or values in variables in order to
  ○ Conveniently refer to them
  ○ Separate Python code from data

```python
>>> greeting = "Welcome to our Introductory Linux Course!"
>>> print(greeting)
Welcome to our Introductory Linux Course!
```
Python Assignment

● Store text or values in variables in order to
  ○ Conveniently refer to them
  ○ Separate Python code from data

```python
>>> greeting = "Welcome to our Introductory Linux Course!"
>>> print(greeting)
Welcome to our Introductory Linux Course!
>>> number = 2
>>> print(number)
2
```
Python Assignment

● Variable names must follow these rules
  ○ No spaces
  ○ Only letters, numbers and _
  ○ Must start with a letter

● Valid names are: my_variable, Value15
● Invalid names are: my-variable, 15th_value
>>> greeting = "Welcome to our Introductory Linux Course!"
>>> print(greeting)
Welcome to our Introductory Linux Course!
>>>
>>> greeting = "Welcome to our Introductory Linux Course!"
>>> print(greeting)
Welcome to our Introductory Linux Course!

>>> number = 2
>>> print(number)
2

>>>
```python
>>> greeting = "Welcome to our Introductory Linux Course!"
>>> print(greeting)
Welcome to our Introductory Linux Course!

>>> number = 2
>>> print(number)
2

>>> print(greeting*number)
Welcome to our Introductory Linux Course!Welcome to our Introductory Linux Course!
```
>>> greeting = "Welcome to our Introductory Linux Course!"
>>> print(greeting)
Welcome to our Introductory Linux Course!

>>> number = 2
>>> print(number)
2

>>> print(greeting*number)
Welcome to our Introductory Linux Course!Welcome to our Introductory Linux Course!

>>> greeting = "Welcome to our Introductory Linux Course!\n"
>>> print(greeting*4)
Welcome to our Introductory Linux Course!
Welcome to our Introductory Linux Course!
Welcome to our Introductory Linux Course!
Welcome to our Introductory Linux Course!

>>>
>>> greeting = "Welcome to our Introductory Linux Course!"
>>> print(greeting)
Welcome to our Introductory Linux Course!

>>> number = 2
>>> print(number)
2

>>> print(greeting*number)
Welcome to our Introductory Linux Course!Welcome to our Introductory Linux Course!

>>> greeting = "Hello!\n"
>>> print(greeting*4)
Hello!
Hello!
Hello!
Hello!

>>>
Python Assignment

Python knows different **types** of data

- **int/float** for numeric data (integer or floating point numbers)
- **str** for text (or sequences of characters, so-called strings)

```python
>>> greeting = "Welcome to our Introductory Linux Course!"
>>> number = 2
>>> print(greeting + number)
```

```output
'Welcome to our Introductory Linux Course!2'
```
Python Assignment

- Python knows different **types** of data
  - **int/float** for numeric data (integer or floating point numbers)
  - **str** for text (or sequences of characters, so-called strings)

```python
>>> greeting = "Welcome to our Introductory Linux Course!"
>>> number = 2
>>> print(greeting + number)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```
Python knows different **types** of data

- **int,float** for numeric data (integer or floating point numbers)
- **str** for text (or sequences of characters, so-called strings)
Python knows different **types** of data

- **int/float** for numeric data (integer or floating point numbers)
- **str** for text (or sequences of characters, so-called strings)

```python
>>> greeting = "Welcome to our Introductory Linux Course!"
>>> number = 2
>>> print(greeting + number)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly

>>> print(number + greeting)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```
Data Types

```python
>>> 4 + 2
6
>>> 
```
Data Types

```python
>>> 4 + 2
6
>>> 4 + 2.0
6.0
>>> 
```
Data Types

>>> 4 + 2
6
>>> 4 + 2.0
6.0
>>> 8 / 5
1.6
>>>  

**Note:** In Python 2 division of two integer numbers results in an integer: \(8/5 = 1\)
Data Types

>>> 4 + 2
6
>>> 4 + 2.0
6.0
>>> 8 / 5
1.6
>>> 8 / 5.0
1.6

**Note:** In Python 2 division of two integer numbers results in an integer: 8/5 = 1
The function `type()` yields information on the type of a variable.
Data Types

- The function `type()` yields information on the type of a variable.

```python
>>> type(8 / 5)
<class 'float'>

>>> a = 8
>>> b = 5
>>> sum = a + b
>>> type(sum)
<class 'int'>
```
The function `type()` yields information on the type of a variable.
Python Strings

- Text, or **strings of characters**, easy to handle
- Given two strings:

\[
A = "ABCD" \quad B = "EFG"
\]

<table>
<thead>
<tr>
<th>Problem</th>
<th>Python</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection of a character</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concatenation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Substring</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prefix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Suffix</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Python Strings

- Text, or *strings of characters*, easy to handle
- Given two strings:

  \[ A = "ABCD" \quad B = "EFG" \]

<table>
<thead>
<tr>
<th>Problem</th>
<th>Python</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection of a character</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concatenation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Substring</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prefix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Suffix</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Note*: Python starts counting from zero!
## Python Strings

- Text, or **strings of characters**, easy to handle
- Given two strings:

  \[
  A = "ABCD" \quad B = "EFG"
  \]

<table>
<thead>
<tr>
<th>Problem</th>
<th>Python</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection of a character</td>
<td>A[0], B[2],</td>
<td>&quot;A&quot;, &quot;G&quot;,</td>
</tr>
<tr>
<td></td>
<td>A[-1]</td>
<td>&quot;D&quot;</td>
</tr>
<tr>
<td>Concatenation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Substring</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prefix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Suffix</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** Python starts counting from zero!
# Python Strings

- Text, or **strings of characters**, easy to handle
- Given two strings:

\[
\begin{align*}
A &= "ABC\text{D}" \\
B &= "EFG"
\end{align*}
\]

<table>
<thead>
<tr>
<th>Problem</th>
<th>Python</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection of a character</td>
<td>A[0], B[2], A[-1]</td>
<td>&quot;A&quot;, &quot;G&quot;, &quot;D&quot;</td>
</tr>
<tr>
<td>Concatenation</td>
<td>B + A</td>
<td>&quot;EFG\text{A}\text{B}\text{C}\text{D}&quot;</td>
</tr>
<tr>
<td>Substring</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prefix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Suffix</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** Python starts counting from zero!
Python Strings

- Text, or **strings of characters**, easy to handle
- Given two strings:

\[
A = "ABCD" \quad B = "EFG"
\]

<table>
<thead>
<tr>
<th>Problem</th>
<th>Python</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection of a character</td>
<td>A[0], B[2], A[-1]</td>
<td>&quot;A&quot;, &quot;G&quot;, &quot;D&quot;</td>
</tr>
<tr>
<td>Concatenation</td>
<td>B + A</td>
<td>&quot;EFGABCD&quot;</td>
</tr>
<tr>
<td>Substring</td>
<td>A[1:3]</td>
<td>&quot;BC&quot;</td>
</tr>
<tr>
<td>Prefix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Suffix</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** Python starts counting from zero!
Python Strings

- Text, or **strings of characters**, easy to handle
- Given two strings:

\[ A = "ABCD" \quad B = "EFG" \]

<table>
<thead>
<tr>
<th>Problem</th>
<th>Python</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection of a character</td>
<td>A[0], B[2], A[-1]</td>
<td>&quot;A&quot;, &quot;G&quot;, &quot;D&quot;</td>
</tr>
<tr>
<td>Concatenation</td>
<td>B + A</td>
<td>&quot;EFGABCD&quot;</td>
</tr>
<tr>
<td>Substring</td>
<td>A[1:3]</td>
<td>&quot;BC&quot;</td>
</tr>
<tr>
<td>Prefix</td>
<td>A[:2]</td>
<td>&quot;AB&quot;</td>
</tr>
<tr>
<td>Suffix</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** Python starts counting from zero!
Python Strings

- Text, or **strings of characters**, easy to handle
- Given two strings:

\[
A = "ABCD" \\
B = "EFG"
\]

<table>
<thead>
<tr>
<th>Problem</th>
<th>Python</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection of a character</td>
<td>A[0], B[2], A[-1]</td>
<td>&quot;A&quot;, &quot;G&quot;, &quot;D&quot;</td>
</tr>
<tr>
<td>Concatenation</td>
<td>B + A</td>
<td>&quot;EFGABCD&quot;</td>
</tr>
<tr>
<td>Substring</td>
<td>A[1:3]</td>
<td>&quot;BC&quot;</td>
</tr>
<tr>
<td>Prefix</td>
<td>A[:2]</td>
<td>&quot;AB&quot;</td>
</tr>
<tr>
<td>Suffix</td>
<td>A[1:], B[-2:]</td>
<td>&quot;BCD&quot;, &quot;FG&quot;</td>
</tr>
</tbody>
</table>

**Note:** Python starts counting from zero!
Methods

```python
>>> greeting = "Welcome!"
>>> dir(greeting)
```
Methods

```python
>>> greeting = "Welcome!"
>>> dir(greeting)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```
Methods

- The function `dir()` yields information on the methods of a specific data type or object

```python
>>> greeting = "Welcome!"
>>> dir(greeting)
[..., 'count', ...]
>>> 
```
Methods

```python
>>> greeting = "Welcome!"
>>> dir(greeting)
[...,'count',...]

>>> help(greeting.count)
Help on built-in function count:

count(...) method of builtins.str instance
    S.count(sub[, start[, end]]) -> int

        Return the number of non-overlapping occurrences of
        substring sub in string S[start:end]. Optional
        arguments start and end are interpreted as in slice
        notation.
```

- The function `help()` yields further information on how to use these methods (press "q" to quit)
>>> greeting = "Welcome!"
>>> dir(greeting)
[...,'count',...]

>>> help(greeting.count)

Help on built-in function count:

count(...)

S.count(sub[, start[, end]]) -> int

Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

>>> greeting.count('e')
2

>>>
Methods

```python
>>> greeting = "Welcome!"
>>> dir(greeting)
[...,'count',...]
>>> help(greeting.count)
Help on built-in function count:

count(...)
S.count(sub[, start[, end]]) -> int

Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

>>> greeting.count('e')
2
>>> greeting.count('e', 2)
1
>>> ```
Methods

```python
>>> greeting = "Welcome!"
>>> dir(greeting)
[...,'count',...]
>>> help(greeting.count)
Help on built-in function count:

count(...)
    S.count(sub[, start[, end]]) -> int

    Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.
```

```python
>>> greeting.count('e')
2
>>> greeting.count('e', 2)
1
>>> greeting.count('e', 2, 4)
0
```
>>> greeting = "Welcome!"
>>> dir(greeting)
[..., 'count', ...]
>>> help(greeting.count)

Help on built-in function count:

count(...)  
S.count(sub[, start[, end]]) -> int

Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

>>> greeting.count('e')
2
>>> greeting.count('e', 2)
1
>>> greeting.count('e', 2, 4)
0
>>> greeting.count('ome')
1
Python Lists

- **Lists** contain series of arbitrary values (to be precise: objects)
- Lists are defined by writing the individual values separated by commas inside square brackets
- One can define empty lists:

  ```python
  >>> l = []
  ```

- Lists can contain different data types

  ```python
  >>> l = [1, 2, "ABC", 7.1]
  ```
Python Lists

- With `append` one can add elements to a list
- Lists can be concatenated with the operator `+`
- Lists can be accessed via index operations in the same way as strings
Python Lists

- With `append` one can add elements to a list
- Lists can be concatenated with the operator `+`
- Lists can be accessed via index operations in the same way as strings

```python
>>> l = []
>>> l.append(0)
>>> l.append(1)
>>> print(l)
[0, 1]
>>> m = [2, 3, 4]
>>> n = l + m
>>> print(n)
[0, 1, 2, 3, 4]
>>> 
```
Python Lists

```python
>>> l = []
>>> l.append(0)
>>> l.append(1)
>>> print(l)
[0, 1]
>>> m = [2, 3, 4]
>>> n = l + m
>>> print(n)
[0, 1, 2, 3, 4]
>>> print(n[2])
2
```

- With **append** one can add elements to a list
- Lists can be concatenated with the operator `+`
- Lists can be accessed via index operations in the same way as strings
Python Lists

- With **append** one can add elements to a list
- Lists can be concatenated with the operator `+`
- Lists can be accessed via index operations in the same way as strings
Python Lists

- With **append** one can add elements to a list
- Lists can be concatenated with the operator **+**
- Lists can be accessed via index operations in the same way as strings
Python Lists

- With `append` one can add elements to a list
- Lists can be concatenated with the operator `+`
- Lists can be accessed via index operations in the same way as strings

```python
>>> l = []
>>> l.append(0)
>>> l.append(1)
>>> print(l)
[0, 1]
>>> m = [2, 3, 4]
>>> n = l + m
>>> print(n)
[0, 1, 2, 3, 4]
>>> print(n[2])
2
>>> l.append([5, 6, 7])
>>> l
... [0, 1, 2, 3, 4, [5, 6, 7]]
>>> l[5]
... [5, 6, 7]
>>> m
[2, 3, 4]
>>> l[5][2]
... 7
>>> n
... [0, 1, 2, 3, 4, [5, 6, 7]]
```
Python Lists

- Python has more useful built-in functions

```python
>>> l = [170, 50, 3, 244]
>>> print(min(l))
3
>>> 
```
Python Lists

- Python has more useful built-in functions

```python
>>> l = [170, 50, 3, 244]
>>> print(min(l))
3
>>> print(max(l))
244
>>> 
```
Python Lists

- Python has more useful built-in functions

```python
>>> l = [170, 50, 3, 244]
>>> print(min(l))
3
>>> print(max(l))
244
>>> print(sorted(l))
[3, 50, 170, 244]
```
Python Lists

- Python has more useful built-in functions

```python
>>> l = [170, 50, 3, 244]
>>> print(min(l))
3
>>> print(max(l))
244
>>> print(sorted(l))
[3, 50, 170, 244]
>>> print(sum(l))
467
```
Python Lists

- Python has more useful built-in functions

```python
>>> l = [170, 50, 3, 244]
>>> print(min(l))
3
>>> print(max(l))
244
>>> print(sorted(l))
[3, 50, 170, 244]
>>> print(sum(l))
467
>>> print(len(l))
4
```
Python Dictionary

● Dictionaries (dict) uses keys instead of index numbers, which makes them unordered
● Dicts are defined by writing key-value pairs separated by commas inside curly brackets
● One can define empty dicts:

```python
>>> D = {}
```

● Dicts can contain different data types

```python
>>> D = {"key1":5, "nextKey":"look, a string", "car":"vw"}
```
You add elements by assigning a value to a key

```python
>>> D = {}
>>> D['start'] = "ABC"
>>> D['end'] = "XYZ"
>>> D['length'] = 26
```
You add elements by assigning a value to a key.

```python
>>> D = {}
>>> D[‘start’] = “ABC”
>>> D[‘end’] = “XYZ”
>>> D[‘length’] = 26
>>> print(D)
{‘length’: 26, ‘end’: ‘XYZ’, ‘start’: ‘ABC’}
>>> 
```
You add elements by assigning a value to a key

```python
>>> D = {}
>>> D['start'] = "ABC"
>>> D['end'] = "XYZ"
>>> D['length'] = 26
>>> print(D)
{'length': 26, 'end': 'XYZ', 'start': 'ABC'}

>>> print(D['start'] + D['end'])
ABCXYZ
```
You add elements by assigning a value to a key.

```python
>>> D = {}
>>> D['start'] = "ABC"
>>> D['end'] = "XYZ"
>>> D['length'] = 26
>>> print(D)
{'length': 26, 'end': 'XYZ', 'start': 'ABC'}

>>> print(D['start'] + D['end'])
ABCXYZ

>>> D['aList'] = [19, 83, 1, 15]
>>> print(D['aList'][3])
15
```
Control Structures

- Control structures determine the logical flow of a program
- There are two types of key control structures in Python:
  - Conditions: **if-else**
  - Loops: **for, while**
- These two types of control structures permit the modeling of all possible program flows
Python Conditions

- Conditions limit the execution of parts of the program
- In Python this construct is called if-else

```python
>>> A = "ACGT"
>>> B = "AAT"
>>> if len(A) < len(B):
...     print("Sequence A is smaller than B.")
```

- If the condition is fulfilled the block following if is executed
Python Conditions

- Conditions limit the execution of parts of the program
- In Python this construct is called if-else

```python
>>> A = "ACGT"
>>> B = "AAT"
>>> if len(A) < len(B):
...     print("Sequence A is smaller than B.")
... else:
...     print("Sequence A is greater or equal than B.")
```

- If the condition is fulfilled the block following if is executed, otherwise the block after else is executed
Python Conditions

- Conditions limit the execution of parts of the program
- In Python this construct is called if-else

```python
>>> A = "ACGT"
>>> B = "AAT"
>>> if len(A) < len(B):
...     print("Sequence A is smaller than B.")
... else:
...     print("Sequence A is greater or equal than B.")
```

- For comparisons different operators are used
  
  ```
  ==   equal
  !=   not equal
  <    less
  >    greater
  <=   equal to or less
  >=   greater or equal
  ```
Python Conditions

- The if-else condition can be extended with elif to consider multiple conditions (as many as necessary)

```python
>>> A = "ABCD"
>>> if len(A) <= 3:
...     print("Sequence A is smaller or equal than 3.")
... ...
```
The if-else condition can be extended with elif to consider multiple conditions (as many as necessary)

```python
>>> A = "ABCD"
>>> if len(A) <= 3:
...     print("Sequence A is smaller or equal than 3."")
...     ...
...     else if (len(A) > 3) and (len(A) < 5):
...     print ("Sequence A is greater than 3 and smaller than 5."")
...     ...
```
Python Conditions

● The if-else condition can be extended with elif to consider multiple conditions (as many as necessary)

```python
>>> A = "ABCD"
>>> if len(A) <= 3:
...     print("Sequence A is smaller or equal than 3.")
... 
... elif (len(A) > 3) and (len(A) < 5):
...     print("Sequence A is greater than 3 and smaller than 5.")
... 
... elif len(A) == 5:
...     print("Sequence A is equal to 5.")
... 
```
Python Conditions

- The if-else condition can be extended with elif to consider multiple conditions (as many as necessary)

```python
>>> A = "ABCD"
>>> if len(A) <= 3:
...     print("Sequence A is smaller or equal than 3."")
... elif (len(A) > 3) and (len(A) < 5):
...     print ("Sequence A is greater than 3 and smaller than 5.")
... elif len(A) == 5:
...     print("Sequence A is equal to 5.")
... else:
...     print("Sequence A is greater than 5.")
```
Note: Assignment and Comparison

- **Assignment:**
  
  store a value in a variable

  ```
  >>> A = "ACGT"
  >>> B = "AAT"
  >>> number = 5
  >>> sum = 10
  ```

- **Comparison:**
  
  compare two values

  ```
  >>> if len(A) == len(B):
  ...     print("Sequence", A, "is equal to sequence", B)
  ... else:
  ...     print("Sequence", A, "is not equal to sequence", B)
  ...
  Sequence ACGT is not equal to sequence AAT
  ```
Python Loops

● For every element of a list (e.g., list, string)
● The head defines the loop variable
● The body is executed for each of the values of the loop variable

```python
>>> l = [170, 50, 3, 244]  # list
>>> for i in l:            # head with loop variable i
...   print(i)            # body
...  # press ENTER a second time
```
Python Loops

- For every element of a list (e.g., list, string)
- The head defines the loop variable
- The body is executed for each of the values of the loop variable

```python
>>> l = [170, 50, 3, 244]  # list
>>> for i in l:
...    print(i)            # head with loop variable i
...                        # body
...                        # press ENTER a second time
170
```

```bash
>>> l = [170, 50, 3, 244]  # list
>>> for i in l:
...    print(i)            # head with loop variable i
...                        # body
...                        # press ENTER a second time
170
```
Python Loops

- For every element of a list (e.g., list, string)
- The head defines the loop variable
- The body is executed for each of the values of the loop variable

```python
>>> l = [170, 50, 3, 244]  # list
>>> for i in l:
...    print(i)           # head with loop variable i
...    print(i)           # body
...                         # press ENTER a second time
170
50
```

```python
>>> 
```
Python Loops

- For every element of a list (e.g., list, string)
- The head defines the loop variable
- The body is executed for each of the values of the loop variable

```python
>>> l = [170, 50, 3, 244]  # list
>>> for i in l:
...     print(i)  # head with loop variable i
...     print(i)  # body
...  # press ENTER a second time
170
50
3
```
Python Loops

- For every element of a list (e.g., list, string)
- The head defines the loop variable
- The body is executed for each of the values of the loop variable

```python
>>> l = [170, 50, 3, 244]  # list
>>> for i in l:
...    print(i)  # head with loop variable i
...    print(i)  # body
...  # press ENTER a second time
170
50
3
244
>>>```
Python Loops

- For every element of a list (e.g., list, string)
- The head defines the loop variable
- The body is executed for each of the values of the loop variable

```python
>>> l = [170, 50, 3, 244]  # list
>>> for my_loop_variable in l:  # head with loop variable i
...     print(my_loop_variable)  # body
... # press ENTER a second time
170
50
3
244
>>>```

>>> l = [170, 50, 3, 244]  # list
>>> for my_loop_variable in l:  # head with loop variable i
...     print(my_loop_variable)  # body
... # press ENTER a second time
170
50
3
244
>>>
Python Loops

- For every element of a list (e.g., list, string)
- The head defines the loop variable
- The body is executed for each of the values of the loop variable

```python
>>> l = [170, 50, 3, 244]  # list
>>> for i in l:            # head with loop variable i
...     print(i)         # body
...                        # press ENTER a second time
```
Python Loops

- For every element of a list (e.g., list, string)
- The head defines the loop variable
- The body is executed for each of the values of the loop variable

```python
>>> l = [170, 50, 3, 244]  # list
>>> for i in l:  # head with loop variable i
...    print(i)  # body
...  # press ENTER a second time
File "<stdin>", line 2
    print(i)  # press ENTER a second time
    ^
IndentationError: expected an indented block
```
Python Loops

- For every element of a list (e.g., list, string)
- The head defines the loop variable
- The body is executed for each of the values of the loop variable

```python
>>> A = "ABCD"  # string
>>> for i in A:  # head with loop variable i
...     print(i)  # body
...  # press ENTER a second time
```
Python Loops

- For every element of a list (e.g., list, string)
- The head defines the loop variable
- The body is executed for each of the values of the loop variable

```python
>>> A = "ABCD"  # string
>>> for i in A:  # head with loop variable i
...    print(i)  # body
...  # press ENTER a second time

>>> A
```
Python Loops

- For every element of a list (e.g., list, string)
- The head defines the loop variable
- The body is executed for each of the values of the loop variable

```python
>>> A = "ABCD"  # string
>>> for i in A:  # head with loop variable i
...     print(i)  # body
... # press ENTER a second time
A
B

>>> `
Python Loops

- For every element of a list (e.g., list, string)
- The head defines the loop variable
- The body is executed for each of the values of the loop variable

```python
>>> A = "ABCD"  # string
>>> for i in A:  # head with loop variable i
...     print(i)  # body
...  # press ENTER a second time
A
B
C
```
Python Loops

- For every element of a list (e.g., list, string)
- The head defines the loop variable
- The body is executed for each of the values of the loop variable

```python
>>> A = "ABCD"  # string
>>> for i in A:  # head with loop variable i
...     print(i)  # body
...  # press ENTER a second time
A
B
C
D
```
Python Loops

- The body may contain multiple lines of code

```python
>>> A = "ABCD"  # string
>>> l = []
>>> for i in A:
...     print(i)  # head with loop variable i
...     l.append(i)  # body
...  # body
...  # press ENTER a second time
A
B
C
D
>>> 
```
Python Loops

- The body may contain multiple lines of code

```python
>>> A = "ABCD"  # string
>>> l = []
>>> for i in A:
...     print(i)  # body
...     l.append(i)  # body
...  # press ENTER a second time
A
B
C
D
>>> print(l)
['A', 'B', 'C', 'D']
>>> 
```
Python Loops

- The build-in function `range()` creates a list of values
- The list elements can be used to iterate through during a for-loop

```python
>>> for i in range(5):
...    print(i)
...    # print(i)
... 0
1
2
3
4
```

**Note:** Python starts counting from zero!
Python Loops

- The build-in function `range()` creates a list of values
- The list elements can be used to iterate through during a for-loop

```python
>>> A = "ABCD"
>>> print(len(A))
4
>>> for i in range(len(A)):
...    print(i)
...
0
1
2
3
>>> 
```
Python Loops

- The build-in function `range()` creates a list of values
- The list elements can be used to iterate through during a for-loop

```python
>>> A = "ABCD"
>>> print(len(A))
4
>>> for i in range(len(A)):
...     print(i, A[i])
...
0 A
1 B
2 C
3 D
>>> for i in A:
...     print(i)
...
A
B
C
D
```
Nested Loops

- The body of the first loop (also called outer loop) includes the body of the second loop (called inner loop)
- Prints all words of length 2 of $\Sigma_{DNA}$:

```python
>>> A = "ACGT"
>>> for i in A:
...     for j in A:
...         print(i + j)
...```
Nested Loops

- The body of the first loop (also called outer loop) includes the body of the second loop (called inner loop)
- Prints all words of length 2 of $\Sigma_{\text{DNA}}$:

```python
>>> A = "ACGT"
>>> for i in A:
...     for j in A:
...         print(i + j)
...```

Python I
Nested Loops

- The body of the first loop (also called outer loop) includes the body of the second loop (called inner loop)
- Prints all words of length 2 of $\Sigma_{\text{DNA}}$: 

```python
>>> A = "ACGT"
>>> for i in A:
...     for j in A:
...         print(i + j)
... AA
```
Nested Loops

- The body of the first loop (also called outer loop) includes the body of the second loop (called inner loop)
- Prints all words of length 2 of $\Sigma_{DNA}$:

```python
>>> A = "ACGT"
>>> for i in A:
...     for j in A:
...         print(i + j)
... AA
AA
AC
```
Nested Loops

- The body of the first loop (also called outer loop) includes the body of the second loop (called inner loop)
- Prints all words of length 2 of $\Sigma_{\text{DNA}}$:

```python
>>> A = "ACGT"
>>> for i in A:           ACGT
...   for j in A:         ACGT
...     print(i + j)
...  
AA
AC
AG
```
Nested Loops

- The body of the first loop (also called outer loop) includes the body of the second loop (called inner loop)
- Prints all words of length 2 of $\Sigma_{DNA}$:

```python
>>> A = "ACGT"
>>> for i in A:
...     for j in A:
...         print(i + j)
...
AA
AC
AG
AT
```
Nested Loops

- The body of the first loop (also called outer loop) includes the body of the second loop (called inner loop)
- Prints all words of length 2 of $\Sigma_{\text{DNA}}$:

```python
>>> A = "ACGT"
>>> for i in A:
...     for j in A:
...         print(i + j)
...     ACGT
...
AA
AC
AG
AT
CA
```
Nested Loops

● The body of the first loop (also called outer loop) includes the body of the second loop (called inner loop)

● Prints all words of length 2 of $\Sigma_{DNA}$:

```python
>>> A = "ACGT"
>>> for i in A:                      ACGT
...     for j in A:                   ACGT
...         print(i + j)
...                  AA
AA
AC
AG
AT
CA
CC
```
Python While-Loops

- While-loops iterate as long as a certain condition is met
- Before each iteration of a while-loop is executed, the condition in the head is tested, while it is still true, the body is executed

```python
>>> A = "ABCD"
>>> i = 0
>>> while i < len(A):
...     print(A[i])
...     i = i + 1

>>> A = "ABCD"
>>> for i in range(len(A)):
...     print(A[i])
```

```python
>>> A = "ABCD"
>>> i = 0
>>> while i < len(A):
...     print(A[i])
...     i = i + 1
```
Writing and Executing a Program

- Python allows not only interactive work, but also the execution of a full program which is saved as a file.
- It reads the full program and executes each line consecutively, starting with the first.
- This program calculates the product of two numbers.

```python
x = 15
print("x = ", x)
y = 456
print("y = ", y)
print("Product of x and y is ", x*y)
```
Writing and Executing a Program

- Use an arbitrary editor (gedit, nano, emacs, vi, vscode, sublime text, ...) to write the program and save it as a file, named e.g. "product.py"

```python
x = 15
print("x = ", x)
y = 456
print("y = ", y)
print("Product of x and y is", x*y)
```

[dahlo@rackham1 ~]$ gedit product.py &
Writing and Executing a Program

- Python-Interpreter vs. editor written program

```python
>>> A = "ACGT"
>>> if len(A) <= 3:
...     print("Sequence A is smaller or equal than 3.")
... elif (len(A) > 3) and (len(A) < 5):
...     print("Sequence A is greater than 3 and smaller than 5.")
... elif len(A) == 5:
...     print("Sequence A is equal to 5.")
... else:
...     print("Sequence A is greater than 5.")
```
Writing and Executing a Program

- To execute the program we call the interpreter from the command line using the filename as argument

[dahlo@rackham1 ~]$ python3 sequence.py
Sequence A is greater than 3 and smaller than 5.
References

- [http://www.diveintopython.net](http://www.diveintopython.net)
  - A full book about Python freely available for download
- [http://openbookproject.net/thinkcs/python/english2e/](http://openbookproject.net/thinkcs/python/english2e/)
  - “How to think like a computer scientist“
  - With examples in Python!

- More information on [http://python.org](http://python.org)

- For example:
  - A Python Tutorial: [https://docs.python.org/2/tutorial/](https://docs.python.org/2/tutorial/)
  - [https://www.codecademy.com/learn/learn-python](https://www.codecademy.com/learn/learn-python)

- You can easily install python on your own computer:
  - [http://www.python.org](http://www.python.org)

- Integrated development environment for python (IDLE)
  - [https://wiki.python.org/moin/IDLE](https://wiki.python.org/moin/IDLE) or [https://code.visualstudio.com/](https://code.visualstudio.com/)
Gedit tips

```python
A = "ACGT"
if len(A) <= 3:
    print("Sequence A is smaller or equal than 3.")
elif (len(A) > 3) and (len(A) < 5):
    print("Sequence A is greater than 3 and smaller than 5.")
elif len(A) == 5:
    print("Sequence A is equal to 5.")
else:
    print("Sequence A is greater than 5.")
```
Gedit tips

```python
A = "ACGT"
if len(A) <= 3:
    print("Sequence A is smaller or equal than 3.")
elif len(A) > 3 and (len(A) < 5):
    print("Sequence A is greater than 3 and smaller than 5.")
elif len(A) == 5:
    print("Sequence A is equal to 5.")
else:
    print("Sequence A is greater than 5.")
```
Gedit tips

```python
A = "ACGT"
if len(A) <= 3:
    print("Sequence A is smaller or equal than 3.")
elif (len(A) > 3) and (len(A) < 5):
    print("Sequence A is greater than 3 and smaller than 5.")
elif len(A) == 5:
    print("Sequence A is equal to 5.")
else:
    print("Sequence A is greater than 5.")
```
Gedit tips

```python
A = "ACGT"
if len(A) <= 3:
    print("Sequence A is smaller or equal than 3.")
elif (len(A) > 3) and (len(A) < 5):
    print("Sequence A is greater than 3 and smaller than 5.")
elif len(A) == 5:
    print("Sequence A is equal to 5.")
else:
    print("Sequence A is greater than 5.")
```
Gedit tips

```python
A = "ACGT"
if len(A) <= 3:
    print("Sequence A is smaller or equal than 3.")
elif (len(A) > 3) and (len(A) < 5):
    print("Sequence A is greater than 3 and smaller than 5.")
elif len(A) == 5:
    print("Sequence A is equal to 5.")
else:
    print("Sequence A is greater than 5.")
```
Gedit tips

● Menu - Preferences - View
  ○ Display line numbers
  ○ Display overview map
  ○ Highlight current line
  ○ Highlight matching brackets

● Menu - Preferences - Editor
  ○ Tab width 4
  ○ Insert spaces instead of tabs

● Menu - Preferences - Fonts & Colors
  ○ Kate or Oblivion (or Monokai \textsuperscript{installed manually, google it})