301AA - Advanced Programming

Lecturer: **Andrea Corradini**

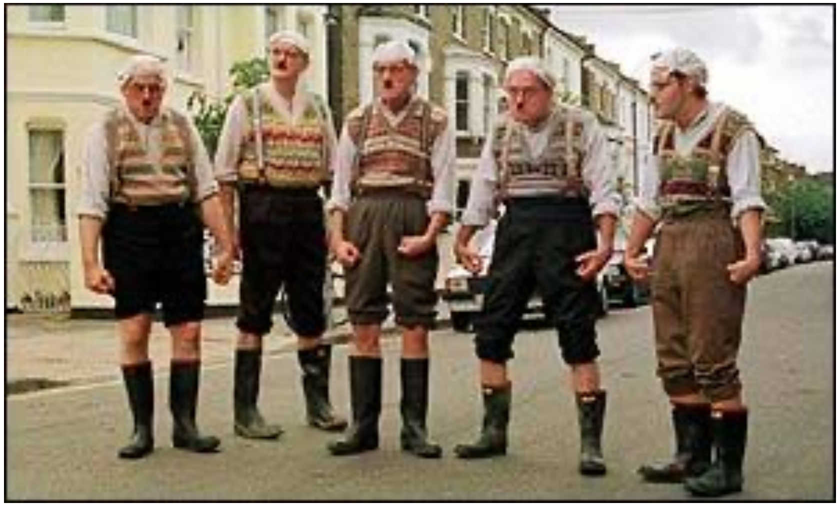
andrea@di.unipi.it

http://pages.di.unipi.it/corradini/

AP-26: *Introduction to Python*

Slides freely adapted from: “Full Python Tutorial”

- **Python Developed by Guido van Rossum in the early 1990s**
  - In July 2018, Van Rossum stepped down as the leader in the language community after 30 years.
- **Named after Monty Python**
- **Available for download from** http://www.python.org



Language features

- Interpreted
- Dynamically typed
- Object oriented (simple object system)
- Supports imperative and functional paradigms
- Several sequence types
  - Strings; List, mutable; Tuples, immutable; Sets
  - Dictionaries (hash maps)
- Powerful subscripting (slicing)
- Higher-order functions (@decorators)
- Flexible signatures
- Iterators and generators
- Exceptions as in Java
- Supports multi-threading
- Indentation instead of braces ({…})

Pragmatics: Why Python?

- Most used general purpose language
- Better Machine Learning libraries!
- Very good example of *scripting, “glue” language*
- “Pythonic” style is very concise
- Powerful but unobtrusive object system
  - *Every* value is an object
- Powerful collection and iteration abstractions
  - Dynamic typing makes generics easy

Dynamic typing – the key difference

- **Java & others: *statically typed***
  - Variable declaration (or type inference) fixes the type
- **Python**
  - Variables come into existence when first assigned to
  - Variables are not typed: Values are typed!
  - A variable can refer to an object of any type
    - Even to objects of different types in the same execution
  - **Strongly typed**: value type does not change in unexpected ways
  - **Type-safe**: no conversion or operation can be applied to values of wrong type
    - *Really?* Not proved… and Bools…
  - Clearly, type errors are only caught at runtime
  - Duck typing (vs. traits and type classes)

“Pythonic” style is very concise

Suggested reading:

- **PEP 8- Style Guide for Python Code**
  - http://www.python.org/dev/peps/pep-0008/
  - The official style guide to Python, contains many helpful programming tips
- **Concise syntax, avoid top-level declarations**

```
class Hello { // Java
        public static void main(String[] args) {
            System.out.println("Hello, world!");
        }
}
---------------------------------------------------
print "Hello, world!\n" # Python
```

- **Python 2.7** supported till 1/1/2020. Now **Python 3.12**

Useful commands of Python interpreter

- Download it from https://www.python.org/
- Current version: 3.12.0
- **help()** Enters Python interactive help utility
- **help(arg)** Prints documentation about **arg**
  - Example: **help(1), help(str), help({}), help(print), help(builtins)**
- **type(arg)** Prints the type of **arg**
  - Example: **type(1), type("Hello"), type(str), type(type)**
- **_ :** in the interpreter is the value of the last expression
- Since "everything is an object", try "dot-completion" to see what are the options…
  - Example: **1. <tab><tab> "hello". <tab><tab>**
  - NB: the latter might not work. Try: "hello" <ret>; _. <tab><tab>

The dir() Function

- The built-in function **dir()** returns a sorted list of strings containing all names defined in a module, a class, or an object

```
>>> import sys
>>> dir(sys)          # Prints names defined in sys
['__displayhook__', '__doc__', '__excepthook__', '__loader__',
'__name__', '__package__', '__stderr__', '__stdin__',
      ...
>>> dir() # Prints names defined currently
      ...
>>> import builtins
>>> dir(builtins) #Prints built-in functions and variables

>>> dir(str) #Prints all members of class str
```

Defining Modules

- **Modules** are files containing definitions and statements. A module defines a new namespace.
- Modules can be organized hierarchically in **packages**

```
# File fibo.py - Fibonacci numbers module
def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Importing a module

```
>>> import fibo    # imports module from local file
'fibo.py'
>>> fibo.fib(6)    # dot notation
[1, 1, 2, 3, 5]
>>> fibo.__name__ # special attribute __name__
'fibo'
>>> fibo.fib.__module__    # special attribute __module__
'fibo'
```

Selective import

```
>>> from fibo import fib, fib2
      # or from fibo import *
>>> fib(500)
>>> fib.__module__            # special attribute __module__
'fibo'
>>> fibo.__name__ # NameError: name 'fibo' is not defined
```

Executing a module as a script

- A module can be invoked as a script from the shell as

```
> python fibo.py 60
```

- Executed with __name__ set to "__main__".

```
# File fibo.py - Fibonacci numbers module
def fib(n):     # write Fibonacci series up to n
    ...
def fib2(n):    # return Fibonacci series up to n
    ...
if __name__ == "__main__": # added code
    import sys
    fib(int(sys.argv[1]))
```

```
> python fibo.py 60
1 1 2 3 5 8 13 21 34
>
```

Basics of Python

```
public class Hello {   // Java
        public static void main(String[] args) {
                // print to the console
                System.out.println("Hello, world");
    }
}
```

```
def main(args):                         # Python
        # print to the console
        print('Hello, world')
```

- Don’t bother with a class unless you actually want to make an object
- Functions don't need return or parameter types
- Indentations matter, not { }.
- Begin functions with : and end by unindenting
- Strings can be " " or ' ', comments begin with #
- No semicolons needed

Basic data types and operators

- **Unbounded integers**
- **Floating point numbers: 64 bits**
- **For numbers `+ - * / %` as expected. `//` int division.**
  - Special use of **%** for string formatting (as with **printf** in C)
- **Logical operators are words (`and, or, not`),** ***not*** **symbols**
- **Strings enclosed in '_', "_", """_"""**
  - `+` also for string concatenation.
- **EOL-comments: `# ...`**
- **Docstrings:**

```
def my_function(x, y):
  """This is the docstring. This
  function does blah blah blah. """
# The code would go here...
```

Assignment

- Assignment in Python does not create a copy
- It sets the name to hold a reference to some object.
- A variable is created *the first time* it appears on the left side of an assignment expression:

  `x = 3`

- An object is deleted (by the garbage collector) once it becomes unreachable.
- **CPython** uses **Reference Counting + Mark & Sweep** for garbage collection
- Multiple assignment:

```
>>> x, y = 2, 3
>>> x
2
>>> y
3
```

Sequence Types

1. **Tuples:** immutable, ordered, heterogeneous
   - Syntax: `()`, `(2, 3.14, False)`, `((2,3), [], "ljshdb")`
2. **Strings (str):** immutable, ordered, only chars (***UTF-8 Unicode***)
3. **Lists :** mutable, ordered, heterogeneous
   - Syntax: `[]`, `[2, 3.14, False]`, `[[2,3], (), "ljshdb"]`

- **Use `list(_)` and `tuple(_)` for conversion**
- **Element selector: <seq>[<index>]**
  - 0 based
  - Negative index start from right (-1)
    - `[1,2,3][0] == 1`   `[1,2,3][-2] == 2`

Operators on sequences

- **Slicing:** returns a subsequence of the original sequence, **a copy**. Start copying at the first index, and stop copying *before* the second index.

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[1:4]        #                                  ('abc', 4.56, (2,3))
>>> t[1:-1]       # negative indices                 ('abc', 4.56, (2,3))
>>> t[1:-1:2]     # optional argument: step          ('abc', (2,3))
>>> t[:2]         # no first index: from beginning   (23, 'abc')
>>> t[2:]         # no second index: to end          (4.56, (2,3), 'def')
>>> t[:]          # no indexes: creates a copy       (23, 'abc', 4.56, (2,3), 'def')
```

- **Concatenation: +** also for tuples and lists: **new** sequence
- **Membership: in** operator

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

Operators on lists only

- **Only lists are mutable:** we can change them *in place.*

```
>>> li = ['abc', 23, 4.34, 23]
>>> li[1] = 45
>>> li
['abc', 45, 4.34, 23]
```

- **append** and **insert**

```
>>> li = [1, 11, 3, 4, 5]
>>> li.append('a')      # Note the method syntax
>>> li
[1, 11, 3, 4, 5, 'a']
>>> li.insert(2, 'i')
>>> li
[1, 11, 'i', 3, 4, 5, 'a']
```

- **extend:** like +, but it adds elements in place
- **index, count:** first occurrence / number of occs [also tuples]
- **remove, reverse, sort, …**

List Comprehensions

```
>>> li = [3, 6, 2, 7]
>>> [elem*2 for elem in li]
[6, 12, 4, 14]
```

[expression for name in list]

- Where **expression** is some calculation or operation acting upon the variable **name**.
- For each member of the list, the list comprehension
  1. sets **name** equal to that member, and
  2. calculates a new value using **expression**,
- It then collects these new values into a list which is the return value of the list comprehension.

[expression for name in list]

List Comprehensions 2

- If the elements of **list** are other collections, then **name** can be replaced by a *collection* of names that match the "shape" of the **list** members.

```
>>> li = [('a', 1), ('b', 2), ('c', 7)]
>>> [ n * 3 for (x, n) in li]
[3, 6, 21]
```

- Sort of pattern matching, also possible for plain assignment…
- Try:

```
>>> (x, y)= (2, 3)
>>> [x, y]= [2, 3]
>>> (x, y)= [2, 3]
>>> (x, y)= "23"
```

Filtered List Comprehension

[expression for name in list if filter]

- **Filter** determines whether **expression** is performed on each member of the list.
- When processing each element of list, first check if it satisfies the filter condition.
- If the filter condition returns *False*, that element is omitted from the list before the list comprehension is evaluated.

```
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem * 2 for elem in li if elem > 4]
[12, 14, 18]
```

- Only 6, 7, and 9 satisfy the filter condition.
- So, only 12, 14, and 18 are produced.

Nested List Comprehensions

- Since list comprehensions take a list as input and produce a list as output, they are easily nested:

```
>>> li = [3, 2, 4, 1]
>>> [elem*2 for elem in
        [item+1 for item in li] ]
[8, 6, 10, 4]
```

- The inner comprehension produces: [4, 3, 5, 2].
- So, the outer one produces: [8, 6, 10, 4].

Sets

- Empty set: **`set()`**
- Indexing not supported
- Mixed types

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange',
'banana'}
>>> print(basket)           # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False
>>> # Demonstrate set operations on unique letters from two words
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                           # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Dictionaries: Like *maps* in Java

- **Dictionaries store a *mapping* between a set of keys and a set of values.**
  - ***Keys can be of any immutable hashable type***
    - ***cannot contain mutable components***
  - Values can be any type
  - Values and keys can be of different types in a single dictionary
- **You can**
  - define
  - modify
  - view
  - lookup
  - delete

the key-value pairs in the dictionary.

Creating and accessing dictionaries

```
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user']
'bozo'

>>> d['pswd']
1234

>>> d['bozo']

Traceback (innermost last):
  File '<interactive input>' line 1, in ?
KeyError: bozo
```

- Keys must be unique.

```
>>> d1 = {1:7,1:5}
>>> d1
{1: 5}
```

Updating Dictionaries

- Assigning to an existing key changes the value.

```
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user'] = 'clown'
>>> d
{'user':'clown', 'pswd':1234}
```

- Assigning to a non-existing key adds a new pair.

```
>>> d['id'] = 45
>>> d
{'user':'clown', 'id':45, 'pswd':1234}
```

- Dictionaries are unordered
  - New entry might appear anywhere in the output.
- (Dictionaries work by *hashing*)

Removing dictionary entries

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> del d['user']                    # Remove one. Note that del is
                                     # a function.
>>> d
{'p':1234, 'i':34}

>>> d.clear()                        # Remove all.
>>> d
{}
>>>  a=[1,2]
>>> del a[1]                        # (del also works on lists)
>>> a
[1]
```

Useful Accessor Methods

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> list(d.keys())                # List of current keys
['user', 'p', 'i']

>>> list(d.values())              # List of current values.
['bozo', 1234, 34]

>>> list(d.items())           # List of item tuples.
[('user','bozo'), ('p',1234), ('i',34)]

>>> list(d)                # When accessing a dictionary as
                           # a list, the keys are returned
['user', 'p', 'i’]
```

Using dictionaries

Write a program to compute the frequency of the words of a string read from the input. The output should print the words in increasing alphanumerical order.

```
freq = {}   # frequency of words in tex
line = input()
for word in line.split():
    freq[word] = freq.get(word,0)+1

words = list(freq.keys())
words.sort()

for w in words:
    print ("%s:%d" % (w,freq[w]))
```

Boolean expressions

- `True` and `False` only constants
- Other values are treated as equivalent to either `True` or `False` when used in conditionals:
  - `False`: zero, `None`, empty containers
  - `True`: non-zero numbers, non-empty objects
  - See PEP 8 for the most Pythonic ways to compare
- Comparison operators: `==`, `!=`, `<`, `<=`, etc.
  - `X == Y` # X and Y have same value (like Java equals method)
  - `X is Y` # X and Y refer to the exact same object (like Java ==)
- Logical connectives
  - `a and b`   `a or b`   `not a`
- Conditional expressions
  - `x = <true_value> if <condition> else <false_value>`
    # lazy

Control statements: conditional

```
if x == 3:
    print("X equals 3.")
elif x == 2:
    print("X equals 2.")
else:
    print("X equals something else.")
print ("This is outside the 'if'.")
```

Note:

- Use of indentation for blocks
- Colon (:) after boolean expression

while Loops

```
>>> x = 3
>>> while x < 5:
        print (x, "still in the loop")
        x = x + 1
3 still in the loop
4 still in the loop
>>> x = 6
>>> while x < 5:
>>>     print (x, "still in the loop")

>>>
```

- **break** inside a loop to leave the **while** loop entirely.
- **continue** inside a loop stops processing the current iteration and immediately go on to the next one.

assert

- An **`assert`** statement will check to make sure that something is true during the course of a program.
  - If the condition if false, the program throws an exception (**`AssertionError`**)

```
assert(number_of_players < 5)
```

For Loops 1

- **For-each** is Python's *only* form of for loop
- A for loop steps through each of the items in a collection type, or any other type of object which is "**iterable**"

```
for <item> in <collection>:
        <statements>
```

- If <collection> is a list or a tuple, then the loop steps through each element of the sequence.

- If <collection> is a string, then the loop steps through each character of the string.

```
for someChar in "Hello World":
        print(someChar)
```

For Loops 2

```
for <item> in <collection>:
        <statements>
```

- **<item>** can be more complex than a single variable name
- In that case it is matched against the structure of the elements of **<collection>**

```
for (x, y) in [('a',1), ('b',2), ('c',3), ('d',4)]:
        print(x)
```

`For` loops and the `range()` function

- We often want to write a loop where the variables ranges over some sequence of numbers. The **`range()`** function returns **an iterator** producing numbers from 0 up to but not including the number we pass to it.
- **`range(5)`** returns an iterator producing **0, 1, 2, 3, 4.**
- So we can write:

```
for x in range(5):
        print(x)
```

- Variant: **`range(start, stop[,step])`**

Abuse of the `range()` function

- Don't use `range()` to iterate over a sequence solely to have the index and elements available at the same time
- Avoid:

```
for i in range(len(mylist)):
    print(i, mylist[i])
```

- Instead:

```
for (i, item) in enumerate(mylist):
    print(i, item)
```

- This is an example of an ***anti-pattern*** in Python
  - For more, see: http://lignos.org/py_antipatterns/