COMP519
Web Programming
Autumn 2015
A Brief Intro to Python
Python

Python was created in the late 1980s and its implementation was started in December 1989 by Guido van Rossum at CWI in the Netherlands.

The name “Python” was chosen by Van Rossum as a working title because he was in a “slightly irreverant mood” and is a big fan of Monty Python’s Flying Circus.

Python 2.0 was released in October 2000, and Python 3.0 was released in December 2008.

Python is described as a “multi-paradigm programming language”, including support for functional programming, object-oriented programming, design by contract, and logic programming.
Python 2.0 and Python 3.0 are not mutually compatible with each other (i.e. it can be important to know which version you are using for syntactic differences exist between them).

Version 2.6.6 and version 3.4.2 are both available on the CS Departmental servers (as well as some earlier versions). Given the status of the CS web server, we will be using Python version 2 for the CGI programming. Hence, I will be concentrating on Python version 2 syntax.

Python can be run in an interactive fashion, or from the command line in the usual scripting language fashion via an interpreter. Programs can also be compiled so as to run without the need of a Python interpreter.
Finally, for our purposes, we will mainly concentrate on the use of Python in the development of CGI programming, i.e. the development of webpages that will use Python scripts when a user submits information to a server.

But first, a brief introduction to the Python language...
A Basic Program

As mentioned, Python can be run in an interactive environment, or scripts can be executed from the command line (in Unix, for example), if the script informs the operating system where to find the Python interpreter.

A simple program to work in the second fashion is as follows:

```
#!/usr/bin/python
#
# basic01.py
# A simple Python program to do the obvious

print "Hello world!"
```

You can run this from the command line by typing

```
bash$ python basic01.py
```

or

```
bash$ ./basic01.py
```

(if the first line is present, and the file is “executable”).
A Basic Program (cont.)

Every Python script (at least those to run from the command line, or for a CGI program), should begin with the line

```
#!/usr/bin/python
```

(or possibly `#!/usr/local/bin/python`).

This is often referred to as the “hash-bang” line (or “she-bang” line) and points the Unix system to the location of the Python (version 2.6.6) interpreter.

Other versions of Python can be accessed by changing “python” to other names, such as `python3.4` (possibly you must change the directory as well).

**Note:** For CGI programming on the CS Departmental server, we must use Python version 2, and the hash-bang line

```
#!/usr/bin/python
```

Single line comments begin with a `#` character.
Variables

Python variables are dynamically typed, but it is also a strongly typed language.

What this means is that you don’t have to define variable data types for variables, Python automatically “guesses” the data type based on the value it contains.

`s = "Hello!"     # creates a string variable`

This assignment statement creates a string variable, and later you can freely re-assign the variable to be a floating point number with no difficulties (unlike, say, Java and C).

`s = 33.65e26     # will not cause an error, even
                 # though s previously assigned!!`
The meaning of “strongly typed” in Python is that variables are not automatically converted from one type to another. The commented out code is valid in, say, JavaScript, but will cause an error in Python.

```python
s = 12
t = " types"
    # print s + t  # This will cause an error in Python!
print str(s) + t  # Correct way to concatenate the
    # two variables to a string
```

Python has five standard data types:
number, string, list, tuple, dictionary

Numbers themselves come in four types:
int, long, float, complex (the “imaginary” unit is j)
Lists

A Python list is (more-or-less) similar to an array in Java or C. One main difference is that a list can store different data types. The syntax is similar to Java arrays, and the indices are numeric, beginning with 0, and defined using square brackets.

```python
list = [ 'abcd', 12, 3.4e12, 'dave', 'bob', 2+3j]
print list  # Prints the complete list
print list[0]  # Prints the first element
print list[2]  # Prints the third element
print list[2:3]  # Also prints the third element
print list[2:4]  # Prints the third and fourth element
print list[2:]  # Prints list starting from the third element
```

In general, a statement like

```python
print list[i:j]  # for i < j
```

will print out the elements `list[i], list[i+1], ..., list[j-1]` (in square brackets).
List Construction and Iterators

The `range()` function provides a variety of methods for building lists and/or providing an iterator in a `for` loop.

```python
# Assumes Python 2 for the range() function

squares = [x**2 for x in range(10)]
    # creates [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
numbers = list(range(4,15))
    # creates [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
del(numbers[0])
    # deletes first element
del(numbers[2:5])
    # deletes elements with indices 2, 3, and 4
del(numbers[2:])
    # deletes elements from index 2 to the end of the list

for i in range(5):
    print i
    # prints integers from 0 to 4 (on separate lines)
for i in range(0, -10, -2):
    print i
    # prints integers from 0 to -8 by steps of -2
    # (on separate lines)
```
Tuples

A Python tuple is a data type similar in nature to a list, consisting of a sequence of comma-separated values, but enclosed in parentheses.

The main difference is that tuples cannot be updated once assigned (think of them as “read only lists”).

tuple = ('doug', 900, 'jones')
print tuple[2] # Prints third element
print tuple[0:2] # Prints the first and second elements # (in parentheses)
# tuple[1] = 1211.3 # Would cause an error!!!
Dictionaries

A Python dictionary is a kind of hash table. They work like associative arrays or hashes found in Perl. Curly braces enclose a dictionary, and values can be assigned and accessed using square braces. Dictionaries consist of key-value pairs. A dictionary key can be almost any Python type but are usually numbers or strings. Values can be any Python object.

dict = { }  # Defines an empty dictionary
dict[‘first’] = 1
dict[2] = "second"
dict["seven"] = "bob"

tinydict = { "name": "Dave", "age": 23, "height": "191cm" }

print dict[‘first’]
print dict  # Prints the entire dictionary
print dict.keys()  # Prints a list of keys of dict
print dict.values()  # Prints a list of values of dict

Note that dictionaries have no concept of “order” associated with the keys or values.
Operations

Python includes (most of) the usual arithmetic and logical operations, as well as (most of) the usual control structures such as for loops, while loops, if branching constructs, etc.

The syntax will likely differ slightly from other languages.

```python
x = "James Bond"
if x == "James Brown"
    print "I feel good!"
    job = "Godfather of soul"
elif x == "James Bond"
    print "I have a license to kill"
    job = "International troublemaker, er, man of mystery"
else:
    print "I have an identity crisis"
    job = "Unknown"
```
Python Syntax

Statements are terminated by the end of a line. No semicolons are necessary. (You can put multiple statement on one line, by separating them with semicolons.)
Statements are terminated by the end of a line. No semicolons are necessary. (You can put multiple statement on one line, by separating them with semicolons.)

**Blocks are delimited by indentation, not the use of curly braces!**

Statements in the same block must be indented by the same amount.
Statements are terminated by the end of a line. No semicolons are necessary. (You can put multiple statement on one line, by separating them with semicolons.)

Blocks are delimited by indentation, not the use of curly braces!

Statements in the same block must be indented by the same amount.

```python
if x:
    if y:
        statement1
else
    statement2
```

Which if does the else clause belong to?
Statements are terminated by the end of a line. No semicolons are necessary. (You can put multiple statement on one line, by separating them with semicolons.)

**Blocks are delimited by indentation, not the use of curly braces!**

Statements in the same block must be indented by the same amount.

```python
if x:
    if y:
        statement1
else
    statement2
```

Which **if** does the **else** clause belong to?

The first **if** (involving x), as is denoted by the **else** “lining up with” that **if**.
One of the immediately noticeable differences between versions 2 and 3 of Python is the syntax for `print`.

In version 2, no parentheses are required (nor expected), whereas in version 3 the `print` function requires parentheses around whatever arguments are being passed to that function.

There are other syntactic and functional differences, but I will generally avoid going into those differences (partly as I don’t know them all myself).
The file *basic02.py* includes some examples of Python syntax for some of the basic control structures like a *for* loop, a *while* loop, printing variables, an *if* statement, etc.

Example programs written are included on the course website. (These may not correspond exactly to the examples included in the notes, but are similar.)

In particular, have a look at this zipped file.

```python
for letter in 'Python':
    print "Current letter", letter

for i in range(10):
    print i**2 + i + 12
```
File handling

Python includes functions for reading and writing to files.

```python
fobj = open("sample.txt")
    # or use
    # fobj = open("sample.txt","r")
    # to open a file for read-only access
for x in fobj:
    print x,

fobj.close()
```

This will print out all lines in the file, one at a time (the comma at the end of the print statement is to avoid printing an extra new line character at the end of each line).

```python
fobj = open("sample.txt")
for x in fobj:
    print x.upper(),

fobj.close()
```
You can write to, or append to, files.

**Note:** Writing to a file (with the "w" option) will actually overwrite the file with no warning provided by Python if the file already exists.

```python
f1 = open("input.txt", "r")
f2 = open("output.txt", "w")
for x in f1:
    f2.write(x.upper())

f1.close()
f2.close()
```
To append text to a file, you can open it with the "a" option. This will put the file pointer at the end of the file (i.e. new content will be added to the end of the file).

```python
f2 = open("output.txt", "a")
f2.write("The quick brown fox...")
f2.close()
```

As you could expect, there is more information available online with regards to file handling, so I will leave you with these basics to start with. (And there are obviously many websites that will provide you with information, not just the one link provided here.)
Strings are enclosed by single or double quotes (use the same to begin and end a string.)

Individual characters in a string can be accessed using an list(i.e. array)-like syntax.

```python
str = "Hello world!"
print str[3:7] # Prints the substring "lo w"
print str[1]  # Prints the substring "e"
```

There are special “escape sequences” which can be included in strings, such as `\n` (a newline character), `\t` (a tab), and `\xnnn` (where n is in the range 0, . . . , 9, a, . . . , f, A, . . . , F, which represents a hexadecimal number).
Python Strings (cont.)

Raw Strings

```python
str = r"Hello!\n"

# A `raw` string does not treat the backslash as
# a special character. Every character in a raw string
# stays the way you wrote it.

print str  # Will print Hello!\n```

Unicode strings

```python
str = u"Hello world!"
```

Unicode strings allow a more varied set of characters, including lots of foreign language characters.

Readers interested in Unicode strings can pursue more information online. (Python 3 treats all strings as Unicode by default.)
There are lots of built-in string methods. Some of the most useful include (assuming `str` is a string):

- `len(str)` Returns the length of the string
- `str.lower()` Converts all uppercase letters to lowercase
- `str.upper()` Converts all lowercase letters to uppercase
- `str.strip([chars])` (e.g. `str.strip(‘ab’)`) Strips characters from the beginning and end of a string. With no arguments, defaults to whitespace.
- `str.join(sequence)` Returns a string in which the string elements of `sequence` have been joined by the `str` separator.

```python
str = "-"
seq = ("a", "b", "c")
print str.join(seq)  # Prints a-b-c
```
Regular Expressions

A regular expression specifies a set of strings that match it. The Python regular expression module (named `re`) gives you functions that let you check if a particular string matches a given regular expression.

For example, the regular expression `pas` matches many (in fact, an infinite number of) strings, such as “spas”, “pass”, “pasttime”, and other (“more nonsense”) strings such as “sttdfcapaspsascc”, etc.

The concept of regular expressions arose in the 1950s and passed into common usage with the development of Unix text processing utilities such as `ed`, an editor, and `grep`, a filtering program.
Regular expressions (cont.)

Many programming languages support the use of regular expressions as a built-in part of the language (such as Perl and JavaScript), or through a standard library (such as Java, Python, and C++).
Many programming languages support the use of regular expressions as a built-in part of the language (such as Perl and JavaScript), or through a standard library (such as Java, Python, and C++).

So what are they?
Many programming languages support the use of regular expressions as a built-in part of the language (such as Perl and JavaScript), or through a standard library (such as Java, Python, and C++).

So what are they?

A regular expression can be a simple string like
- dogs
- Michelle
- The quick brown fox jumped over the lazy dogs.

A regular expression either matches, or doesn’t match a specific string.
Searching and Matching in Python

Python has two main functions for working with regular expressions, which require the use of the module `re`.

```python
#!/usr/bin/python

import re      # import the relative expression module

line = "I like cats and dogs too."

matchObj = re.match("dogs", line)  # re.match checks for a match at the beginning of a string
matchObj = re.search("dogs", line)  # re.search checks for a match anywhere in the string
```

The general syntax for using one of these functions is

```
re.search(pattern, string, flags)
```

where `pattern` is used to check for a match in `string`, and `flags` are (optional) parameters used to modify the search.
In the context of regular expressions, there are several characters that take on a special meaning. Namely, the characters that are control characters are

( + ? . * ^ $ ( ) [ ] { } | \ )

To include these special characters in a regular expression, you must “escape” them, by using a backslash in front of them.

Some examples of special regular expression constructions are included below:

^ Matches beginning of line.
$ Matches end of line.
. Matches any single character except newline.
[...] Matches any single character in brackets.
[^...] Matches any single character not in brackets.
Special Relative Expression Characters (cont.)

\texttt{re*}  Matches 0 or more occurrences of preceding expression.
\texttt{re+}  Matches 1 or more occurrence of preceding expression.
\texttt{re?}  Matches 0 or 1 occurrence of preceding expression.
\texttt{re\{n\}} Matches exactly \( n \) number of occurrences of preceding expression.
\texttt{re\{n,\}} Matches \( n \) or more occurrences of preceding expression.
\texttt{re\{n, m\}} Matches at least \( n \) and at most \( m \) occurrences of preceding expression.
\texttt{a \mid b} Matches either \( a \) or \( b \).
Special Relative Expression Characters (cont.)

(re) Groups regular expressions and remembers matched text.
\w Matches word characters.
\W Matches nonword characters.
\s Matches whitespace. Equivalent to [\t\n\r\f].
\S Matches nonwhitespace.
\d Matches digits. Equivalent to [0-9].
\D Matches nondigits.
\b Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
\n, \t, etc. Matches newlines, carriage returns, tabs, etc.
Constructing regular expressions

Building regular expressions can be somewhat of an art form, and takes practice to get the meaning right.

Some examples are included below:

```
t.e # t followed by anything followed by e
    # This will match tre, tle but not te, tale
^f # f at the beginning of a line
^ftp # ftp at the beginning of a line
e$ # e at the end of a line
tle$ # tle at the end of a line
und* # un followed by zero or more d characters.
    # This will match un, und, undd, unddd, . . . ,
    # lunch, lunches, brunch, . . .
.* # Any string without a newline. This is because
    # the . matches anything except a newline and
    # the * means zero or more of these.
^$ # A line with nothing in it.
[qjk] # Either q or j or k
[^qjk] # Neither q nor j nor k
[a-z] # Anything from (lower-case) a to z inclusive
[^a-z] # No lower case letters
[a-zA-Z] # Any letter
[a-z]+ # Any non-zero sequence of lower case letters
```
Constructing regular expressions (cont.)

We can use the vertical bar | for an "or" and (...) for grouping things together.

```
jelly|cream    # Either jelly or cream
(eg|le)gs    # Either eggs or legs
(da)+        # Either da or dada or dadada or...
```

To get any of the control characters, we need to precede them by a backslash.

```
\|          # Vertical bar
\[          # An open square bracket
\)          # A closing parenthesis
\*          # An asterisk
\^          # A carat symbol (or circumflex)
\\           # A backslash
```
More regular expression examples

Suppose you were building a “calculator” type of application. When parsing an expression, you might want to check for division by 0.

You could build up these types of regular expressions to do so.

```
\0 # A division by zero: "/0"
/ 0 # A division by zero with a space: "/ 0"
/\s0 # A division by zero with a whitespace:
    # "/ 0" where the space may be a tab etc.
/ *0 # A division by zero with possibly some spaces: "/0" or "/ 0" or "/ 0" etc.
/\s*0 # A division by zero with possibly some whitespace.
/\s*0\.0* # As the previous one, but with decimal point and maybe some 0s after it. Accepts "/0." and "/0.0" and "/0.00" etc and "/ 0." and "/ 0.0" and "/ 0.00" etc.
```
Regular expression usage

Regular expressions are, by default, “greedy” in that they will attempt to match as much as possible.

```python
#!/usr/bin/python
#   basic05.py

import re

line = "I like cats and dogs too."

searchObj = re.search ("d.*", line)

if searchObj:
    print "search --> searchObj.group():", searchObj.group()
else:
    print "No match!!"
```

This will result in

```
d dogs too.
```
Regular expression usage (cont.)

Using parentheses, the matching “engine” can remember patterns. These patterns can be recalled for later processing and/or used again within the same regular expression.

Here is an example that will search for repeated words in a string.

```python
#!/usr/bin/python
# basic06.py

import re

line = "I like like cats and dogs too."

searchObj = re.search(r"(\w+) \1", line)
if searchObj:
    print "search --> searchObj.group(1):", searchObj.group(1)
else:
    print "No match!!"
```

This results in

like
Regular expression usage (cont.)

Note that \1 refers to the string (if any) matched by the (first) set of parentheses in the regular expression. So a search is performed for any “word” (consisting of one or more letters and/or numbers and/or underscores), then one or more whitespaces, and then the word \1 again.
Regular expression usage (cont.)

Note that \1 refers to the string (if any) matched by the (first) set of parentheses in the regular expression. So a search is performed for any “word” (consisting of one or more letters and/or numbers and/or underscores), then one or more whitespaces, and then the word \1 again.

searchObj.group(1) also refers to the string (if any) matched by the first group of parentheses.
Regular expression usage (cont.)

Note that \1 refers to the string (if any) matched by the (first) set of parentheses in the regular expression. So a search is performed for any “word” (consisting of one or more letters and/or numbers and/or underscores), then one or more whitespaces, and then the word \1 again.

searchObj.group(1) also refers to the string (if any) matched by the first group of parentheses.

This will find a repeated word if it occurs twice (or more) separated by one or more whitespaces. If the search string was "I like liked cats and dogs too." this would not match, as the words are not exactly the same.
Regular expression usage (cont.)

Note that \1 refers to the string (if any) matched by the (first) set of parentheses in the regular expression. So a search is performed for any “word” (consisting of one or more letters and/or numbers and/or underscores), then one or more whitespaces, and then the word \1 again.

searchObj.group(1) also refers to the string (if any) matched by the first group of parentheses.

This will find a repeated word if it occurs twice (or more) separated by one or more whitespaces. If the search string was "I like liked cats and dogs too." this would not match, as the words are not exactly the same.

A (lowercase or uppercase) r at the start of the search string makes the string into a “raw” string. Otherwise, the Python interpreter will try to treat the backslashes as special characters. Typically, search strings in Python are expressed using this raw notation.
re.findall(pattern, string [,flags]) returns all non-overlapping matches of pattern in string as a list of strings (scanning from left-to-right). If one or more groups are present in the pattern, then this returns a list of groups, or a list of tuples if the pattern has more than one group. Optional flags could be included, as in other regular expression operations.
re.findall(pattern, string [,flags]) returns all non-overlapping matches of pattern in string as a list of strings (scanning from left-to-right). If one or more groups are present in the pattern, then this returns a list of groups, or a list of tuples if the pattern has more than one group. Optional flags could be included, as in other regular expression operations.

Entire books have been written on the construction and usage of regular expressions.

Interested readers can (as always) find more information online about the construction and use of regular expressions.

A few references have been linked from the course website.
Other regular expression and string operations

`re.sub(pattern, repl, string [,count])`
(need to import the `re` module)
Returns a string obtained by replacing the leftmost non-overlapping occurrences of `pattern` by `repl` in `string`. `count` is the maximum number of occurrences to be replaced. If `count` is 0 or missing, all occurrences will be replaced. Back references can be used, as well as the usual regular expression character escape sequences.

There are other common string operations that could be useful.
Suppose that `str` is a string variable.

`str.split([sep [,maxsplit]])`
Returns a list of words in the string, using `sep` as the delimiter. If `maxsplit` is given, the list will have at most `maxsplit + 1` elements. `str.split()` will split on whitespace.
As expected, you can define your own functions in Python, which can take parameters (or not), and return objects (or not). Functions can be defined that have required arguments, variable-length arguments, default arguments, and functions can be called using keyword arguments.

Variables are called by reference only, so changing the variables inside of a function will result in them being changed when the function returns.

Variables defined inside a function are local in scope, and those defined outside have a global scope.

Similar to other Python syntax, the block of code that defines a function is denoted using the indentation syntax.
def print_this(str):
    print str
    return

def find_min(numList):
    "This function will find the minimum value of a list of numbers."
    if len(numList) == 0:
        return None
    current_min = numList.pop()
    for i in varlist:
        if i < current_min:
            current_min = i
    return current_min

print_this("Here is an example string.")
print find_min([20,-2,33,40])
print find_min([])