Python

Course Map

Batch Scripting

We Are Here

Python

Regular Expressions
&
Wildcards

MakeFiles

Configuration
Management

MS-Excel
Vs.
Google
Spreadsheets

Agenda -1

- Introduction
- How Python Works?
- Seeking Help
- Google Python Class
- Python Setup
- Python Syntax
- Python Editors
- **Lab-1**
- Python Source Code
- Python Native Types
  - Boolean Types
  - Numeric Types
- **Lab-2**
- Sequence Types
  - Strings
- Conditional Statements
  - If…Elif…else
  - Truth Value
- **Lab-3**

Popularity of Programming Language

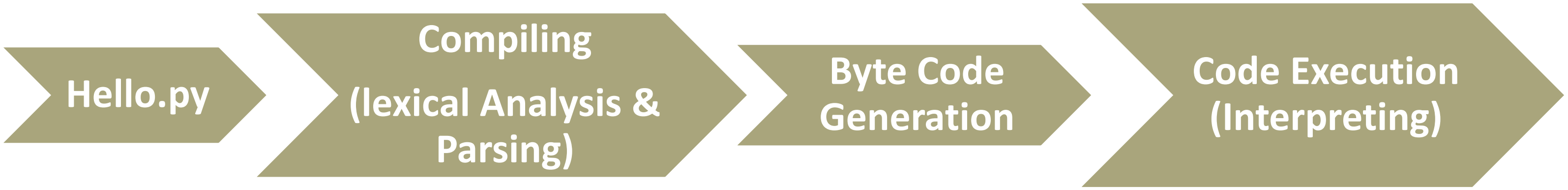
Rank	Language	Share	Trend
1	Python	22.7 %	-0.7 %
2	Java	16.3 %	+3.7 %
3	PHP	8.9 %	-1.1 %
4	C#	8.3 %	-0.5 %
5	Javascript	8.0 %	+0.5 %
6	C++	6.6 %	-0.2 %
7	C	6.4 %	-0.7 %
8	R	3.6 %	+0.4 %
9	Objective-C	3.6 %	-1.2 %
10	Swift	2.8 %	-0.3 %
11	Matlab	2.3 %	-0.2 %
12	Ruby	1.8 %	-0.5 %
13	VBA	1.5 %	+0.0 %
14	Visual Basic	1.4 %	-0.3 %
15	TypeScript	1.3 %	+0.4 %
16	Scala	1.2 %	+0.1 %
17	Perl	0.8 %	-0.3 %
18	Go	0.6 %	+0.2 %

Introduction

- Python is an easy to learn, powerful programming language.
- Ideal language for scripting and rapid application development in many areas on most platforms.
- The Python interpreter and the extensive standard library are **freely available** in **source** or **binary** form for all major platforms
- Most popular implementation of the python interpreter is called **CPython** and is written in **C**.
- Named after the BBC show "**Monty Python's Flying Circus**"

https://www.python.org/

How Python Works?



Seeking Help

- https://docs.python.org/2.7/
- https://wiki.python.org/moin/

Google's Python Class

- This python course will be based on Google's Python Class:
  - https://developers.google.com/edu/python/
- Extra topics & notes will be included in the slides.

Python Set Up

- Go to the [python.org download](https://www.python.org/downloads/) page, select version **2.7.10.** Google's Python Class should work with any version 2.4 or later.
- Run the Python installer, taking all the defaults.
- With Python installed, open a command prompt and run **hello.py**, use the below link to download hello.py
  - https://drive.google.com/file/d/0B6Hf8UvSSqTXTElHX25faHZKRzA/view?usp=sharing

```
C:\google-python-exercises> python hello.py
Hello World
C:\google-python-exercises> python hello.py ITI_39
Hello ITI_39
```

Python Syntax

- Python is **case sensitive** so "**a**" and "**A**" are different variables.
- There are no type declarations of variables, parameters, functions, or methods in source code.
- Python variables don't have any type spelled out in the source code (i.e. There is no **int a=5** it's **a=5**), so it's extra helpful to give meaningful names to your variables to remind yourself of what's going on.

Python Syntax

- A Python program is divided into a number of **logical lines** and there is no **;** to end a statement as in C-language.
- A logical line is constructed from one or more **physical lines** by following the explicit or implicit **line joining** rules.
- A physical line is a sequence of characters terminated by an end-of-line sequence:
  - Unix form using ASCII LF (linefeed – “\n”)
  - Windows form using the ASCII sequence CR LF (return followed by linefeed)
  - Old Macintosh form using the ASCII CR (return - "\r”) character
- A logical line that contains only spaces, tabs, formfeeds and possibly a comment, is **ignored**.

Python Syntax

- **Indentation:**
  - Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the **indentation level** of the line, which in turn is used to determine the grouping of statements(i.e there are no { } as there is in C-language).

```
def foo(grade):
    if grade > 25:
        result = “pass”
    else:
        result = “fail”
    return result
```

```
char foo(int grade)
{
    char result;
    if (grade>25)
    {
      result = 1;
    }
    else
     {
        result = 0;
     }
return result;
}
```

Python Syntax

- **Indentation:**
  - **Cross-platform compatibility note:** it is unwise to use a **mixture** of **spaces** and **tabs** for the indentation in a single source file. It should also be noted that different platforms may explicitly limit the maximum indentation level.

Python Syntax

- **Comments:**
  - A comment starts with a hash character (**#**), and ends at the end of the physical line

```
# This is a single line comment
```

- **Triple-quoted strings** can be used a block line comment.

```
' ' '
This is a multiline
comment.
' ' '
```

```
" " "
This is a multiline
comment.
" " "
```

Python Syntax

- **Line Joining:**
  - Two or more physical lines may be joined into logical lines using backslash characters (\)

```
if 1900 < year < 2100 and 1 <= month <= 12 \
and 1 <= day <= 31 and 0 <= hour < 24 \
and 0 <= minute < 60 and 0 <= second < 60: # Looks like a valid date
    return 1
```

  - A line ending in a backslash cannot carry a comment. A backslash does not continue a comment.

Python Syntax

- **Line Joining:**
  - Expressions in parentheses, square brackets or curly braces can be split over more than one physical line without using backslashes. For example:

```
month_names = ['Januari', 'Februari', 'Maart',      # These are the
               'April', 'Mei', 'Juni',              # Dutch names
               'Juli', 'Augustus', 'September',     # for the months
               'Oktober', 'November', 'December']   # of the year
```

  - Implicitly continued lines can carry comments. The indentation of the continuation lines is not important. Blank continuation lines are allowed.

Python Syntax

- **Syntax Guidelines:**
  - Python Enhancement Proposals (PEP) https://www.python.org/dev/peps/pep-0008/
  - In this course the following syntax guidelines will be used:
    - 2-spaces as the indent
    - Spaces instead of tabs
    - Unix line-ending convention

Python Editors

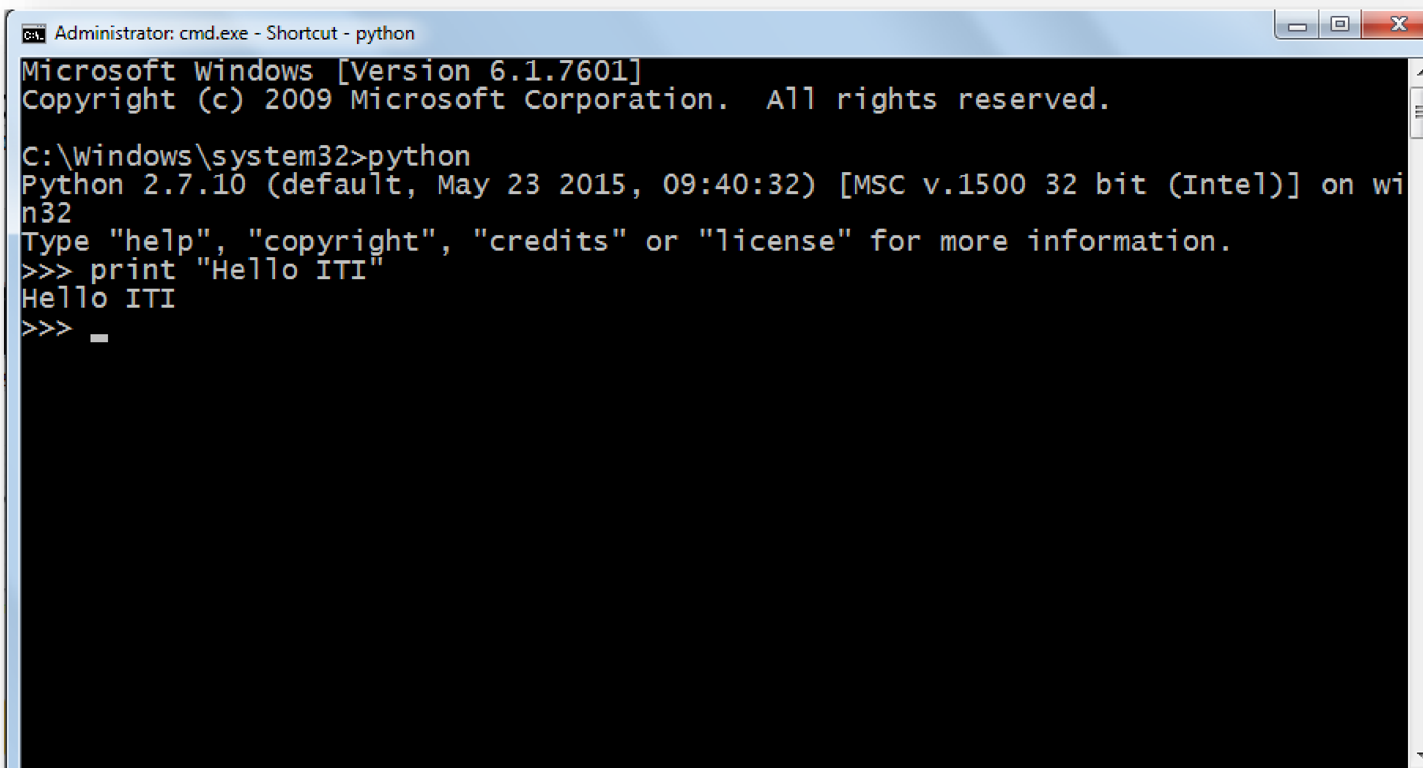
- **Available Editors:**
  - IDLE (C:\Python27\Lib\idlelib\idle.bat)
  - Notepad++ (used in this course)

- **Configuring Notepad++:**
  - Settings > Preferences > Tab settings > tab size=2 and check replace by space
  - Settings > Preferences > MISC > select auto-indent
  - Settings > Preferences > New Document > select Unix/OSX

LAB – 1
TRY PYTHON INTERPRETER

Python Interpreter

- To run the Python interpreter interactively, type "python" in the CMD. On Windows, use **Ctrl-Z** to exit (on all other operating systems it's **Ctrl-D** to exit).



```
$ python         ## Run the Python interpreter
Python 2.7.9 (default, Dec 30 2014, 03:41:42)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-55)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 6        ## set a variable in this interpreter session
>>> a            ## entering an expression prints its value
6
>>> a + 2
8
>>> a = 'hi'     ## 'a' can hold a string just as well
>>> a
'hi'
>>> len(a)       ## call the len() function on a string
2
>>> a + len(a)  ## try something that doesn't work
Traceback (most recent call last):
  File "", line 1, in
TypeError: cannot concatenate 'str' and 'int' objects
>>> a + str(len(a))  ## probably what you really wanted
'hi2'
>>> foo          ## try something else that doesn't work
Traceback (most recent call last):
  File "", line 1, in
NameError: name 'foo' is not defined
>>> ^D           ## type CTRL-d to exit (CTRL-z in Windows/DOS terminal)
```

LAB – 1
TRY PYTHON INTERPRETER

Python Source Code

- Python source files use the "**.py**" extension and are called "**modules** ".
- Here's the hello.py program from the exercises folder.

```
# import modules used here -- sys is a very standard one
import sys

# Gather our code in a main() function
def main():
    print 'Hello there', sys.argv[1]
    # Command line args are in sys.argv[1], sys.argv[2] ...
    # sys.argv[0] is the script name itself and can be ignored

# Standard boilerplate to call the main() function to begin
# the program.
if __name__ == '__main__':
    main()
```

Python Source Code

- **Modules and Imports:**
  - The outermost statements in a Python file, or "module", do its one-time setup. Those statements run from top to bottom the first time the module is imported somewhere, setting up its variables and functions (**Global Space**).
  - A Python module can be run directly "python hello.py Mohammad" or it can be **imported** and used by some other module.
  - When a Python file is run directly, the special variable "**__name__**" is set to "**__main__**". Therefore, it's common to have the boilerplate if **__name__** ==**__main__** as shown in the last example to call a main() function when the module is run directly, but not when the module is imported by some other module.

Python Source Code

- **Modules and Imports:**
  - Suppose you've got a module "**ITI.py**" which contains a "**def foo()**". The fully qualified name of that foo function is "**ITI.foo**". In this way, various Python modules can name their functions and variables whatever they want, and the variable names won't conflict — **module1.foo** is different from **module2.foo**.
  - In the Python vocabulary, we'd say that ITI, module1, and module2 each have their own "**namespaces**".

Python Source Code

- **Modules and Imports:**
  - For example, we have the standard "**sys**" module that contains some standard system facilities, like the **argv** list, and **exit()** function. With the statement "**import sys**" you can then access the definitions in the sys module and makes them available by their fully-qualified name, e.g. **sys.exit()**.

```
import sys
# Now you can refer to sys.xxx facilities
sys.exit(0)
```

  - There is another import form that makes argv and exit() available by their short names

```
from sys import argv, exit
# Now you can refer to exit from sys modules directly
exit(0)
```

Python Source Code

- **Modules and Imports:**
  - There are many modules and packages which are bundled with a standard installation of the Python interpreter.
    - sys — access to exit(), argv, stdin, stdout, ...
    - re — regular expressions
    - os — operating system interface, file system
    - math — mathematical functions defined by the C standard.
    - For a complete list on the available standard modules:
      - https://docs.python.org/2/library/

Self-Study

RESERVED CLASSES OF IDENTIFIERS

Python Source Code

- **<u>Reserved classes of identifiers:</u>**
  - Certain classes of identifiers (besides keywords) have special meanings. These classes are identified by the patterns of leading and trailing underscore characters:
    - _*
      - This is used to indicate that attribute or method is intended to be private. Not imported by **from** module **import** *.
    - __*__
      - System-defined names. These names are defined by the interpreter and its implementation (including the standard library). *Any* use of __*__ names, in any context, that does not follow explicitly documented use, is subject to breakage without warning.

Python Source Code

- **Reserved classes of identifiers:**
  - Certain classes of identifiers (besides keywords) have special meanings. These classes are identified by the patterns of leading and trailing underscore characters:
    - __*
      - Class-private names. Names in this category, when used within the context of a class definition, are re-written to use a **mangled** form to help avoid name clashes between “private” attributes of base and derived classes.
        - Example on mangling: The identifier __spam occurring in a class named Ham will be transformed to _Ham__spam.

Self-Study

RESERVED CLASSES OF IDENTIFIERS

Python Source Code

- **help() and dir():**
  - The **help()** function pulls up documentation strings for various modules, functions, and methods:
    - help(len) — help string for the built-in len() function; note that it's “len” not “len()”.
    - help(sys) — help string for the sys module (must do an import sys first).
  - The **dir()** function tells you what the attributes of an object are:
    - dir(sys) — dir() is like help() but just gives a quick list of its defined symbols, or “attributes ".
    - dir(str) — displays ”str” object attributes, including its methods

Python Source Code

- **User-defined Functions:**
  - Functions in Python are defined like this:

```
# Defines a "repeat" function that takes 2 arguments.
def repeat(s, exclaim):
    """
    Returns the string 's' repeated 3 times.
    If exclaim is true, add exclamation marks.
    """
    result = s + s + s # can also use "s * 3" which is faster (Why?)
    if exclaim:
        result = result + '!!!'
    return result
```

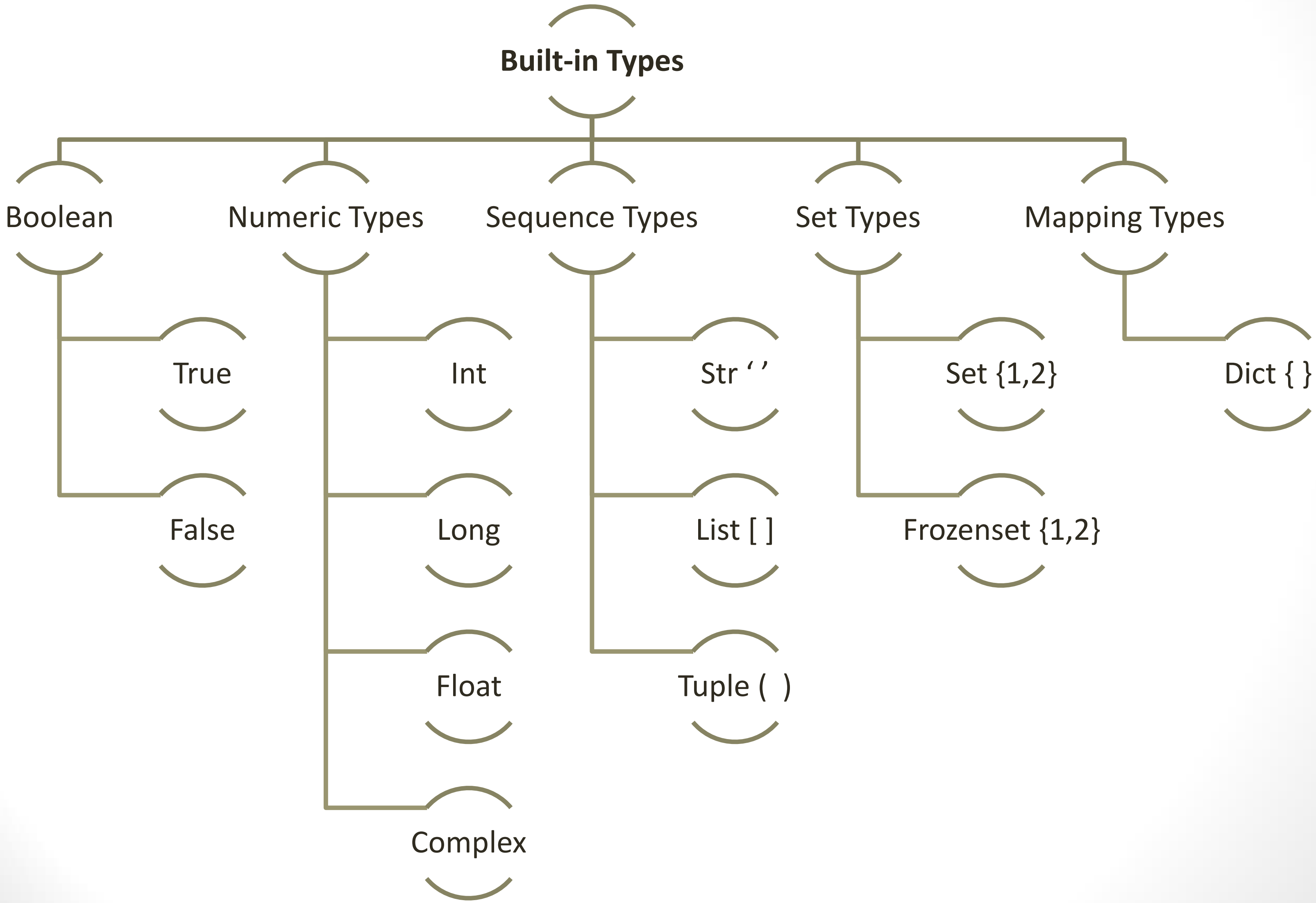
  - The **def** keyword defines the function with its parameters within parentheses and its code indented. The first line of a function can be a documentation string ("**docstring**") that describes what the function does.

Python Source Code

- **User-defined Functions:**
  - Variables defined in the function are **local** to that function, so the "result" in the example function is separate from a "result" variable in another function.
  - The **return** statement can take an argument, in which case that is the value returned to the caller. Here is code that calls the above repeat() function, printing what it returns:

```
def main():
    print repeat('Yay', False)      ## YayYayYay
    print repeat('Woo Hoo', True)   ## Woo HooWoo HooWoo Hoo!!!
```

Python Native Types



Boolean Type

- Boolean type consists of only two possible values:
  - **True** which is equal to **1**
  - **False** which is equal to **0**

- Python's Booleans were added with the primary goal of making code clearer.

```
def isemployee(name):
    for  emp in company_emps:
        if name==emp:
            return 1
        else:
            return 0
```

```
def isemployee(name):
    for  emp in company_emps:
        if name==emp:
            return True
        else:
            return False
```

Numeric Types

- There are four distinct numeric types:
  - Integers
    - Implemented using **long** in C.
    - At least 32 bits of precision.
    - Use **sys.maxint** & **-sys.maxint - 1** to know **max** & **min** values of integer on your machine .
  - Long integers
    - Unlimited precision.
  - Floating point numbers
    - Implemented using **double** in C.
    - Use **sys.float_info** for information about precision and internal representation on your machine.
  - Complex numbers.
    - Complex numbers have a real and imaginary part, which are each a floating point number.

Numeric Types

- To produce numbers of a specific type:
  - Appending '**L**' or '**l**' to a number yields a long integer
    - my_long_int = 9L
  - Numeric literals containing a decimal point or an exponent sign yield floating point numbers.
    - my_fpoint_num=1e+1 (= 10.0)
  - Appending ‘**J**' or ‘**j**' to a number yields a complex number
    - my_complex_num=5+9J
  - The constructors **int()**, **long()**, **float()**, and **complex()** can be used to produce numbers of a specific type.

LAB – 2
NUMERIC TYPES

Exercise on Numeric Types

- Using python interpreter try the following

```
17 / 3        # int / int -> int
17 / 3.0      # int / float -> float
17 // 3.0     # explicit floor division discards the fractional part
17 % 3        # the % operator returns the remainder of the division
5 * 3 + 2
5 ** 2        # 5 squared
2 ** 7        # 2 to the power of 7
```

- Try the following methods (type **help(**method name**)** for help):
  - abs(x)   # try it with a complex number
  - divmod(x, y)
  - pow(x, y)
  - math.floor(x)   # import math to use
  - math.ceil(x)   # import math to use

LAB – 2
NUMERIC TYPES

Sequence Types
Strings

- Python has a built-in string class named "**str**" with many handy features.
- String literals can be enclosed by either double or single quotes (“Hello” or ‘Hello’).
- Python strings are "**immutable**" which means they cannot be changed after they are created (Java strings also use this immutable style).
- Since strings can't be changed, we construct ***new*** strings as we go to represent computed values. So for example the expression ('**hello**' + '**there**') takes in the **2** strings 'hello' and 'there' and builds a **new** string '**hellothere**'.

Sequence Types
Strings

- Python uses **zero-based** indexing, so if mystr = 'hello', then mystr[1] is 'e'. If the index is out of bounds for the string, Python raises an error.
- The **len**(mystr) function returns the length of a string.
- The [ ] syntax and the len() function actually work on any sequence type -- strings, lists, etc.
- The '+' operator can concatenate two strings and the '*' is the repeat operator.

```
s = 'hi‘
print s + ' there'  ## hi there
print ‘-’ * 5 # -----
pi = 3.14
##text = 'The value of pi is ' + pi      ## NO, does not work
text = 'The value of pi is '  + str(pi)  ## yes
```

Sequence Types
Strings

- The "**print**" operator prints out one or more python items followed by a newline (leave a trailing comma at the end of the items to inhibit the newline).

- A "**raw**" string literal is prefixed by an '**r**' and passes all the chars through without special treatment of backslashes

```
raw = r'this\n\t and that'
print raw     ## this\t\n and that
not_raw='this\n\t and that'
print not_raw   ## this
                ##    and that
```

Sequence Types
Strings

- **String Slices:**
  - The "slice" syntax is a handy way to refer to sub-parts of sequences -- typically strings and lists. The slice **s**[**start**:**end**] is the elements beginning at start and extending up to but not including end. the Suppose we have **s** = "**Hello**"

```
 H   e   l   l   o
 0   1   2   3   4
-5  -4  -3  -2  -1
```

  - s[1:4] is 'ell' -- chars starting at index 1 and extending up to but not including index 4
  - s[1:] is 'ello' -- omitting either index defaults to the start or end of the string

Sequence Types
Strings

- **String Slices:**

```
 H   e   l   l   o
 0   1   2   3   4
-5  -4  -3  -2  -1
```

  - s[:] is 'Hello' -- omitting both always gives us a copy of the whole thing (this is the pythonic way to copy a sequence like a string or list)
  - s[1:100] is 'ello' -- an index that is too big is truncated down to the string length
  - s[-1] is 'o' -- last char (1st from the end)
  - s[:-3] is 'He' -- going up to but not including the last 3 chars.
  - s[-3:] is 'llo' -- starting with the 3rd char from the end and extending to the end of the string.

Sequence Types
Strings

- **String % :**
  - Python has a **printf**()-**like** facility to put together a string. The **%** operator takes a printf-type format string on the left (**%d** int, **%s** string, **%f**/**%g** floating point), and the matching values in a tuple on the right (a tuple is made of values separated by commas, typically grouped inside parenthesis)

```
# % operator
number = 1
string = “ITIans"
text = “Hello %s this is an integer  %d" % (string, number)
```

Sequence Types
Strings

- **String Methods:**
  - s.lower(), s.upper() -- returns the lowercase or uppercase version of the string
  - s.strip() -- returns a string with whitespace removed from the start and end
  - s.isalpha()/s.isdigit()/s.isspace()... -- tests if all the string chars are in the various character classes
  - s.startswith('other'), s.endswith('other') -- tests if the string starts or ends with the given other string
  - s.find('other') -- searches for the given other string (not a regular expression) within s, and returns the first index where it begins or -1 if not found

Sequence Types
Strings

- **String Methods:**
  - s.replace('old', 'new') -- returns a string where all occurrences of 'old' have been replaced by 'new‘
  - s.split('delim') -- returns a list of substrings separated by the given delimiter. The delimiter is not a regular expression, it's just text. 'aaa,bbb,ccc'.split(',') -> ['aaa', 'bbb', 'ccc']. As a convenient special case s.split() (with no arguments) splits on all whitespace chars.
  - s.join(list) -- opposite of split(), joins the elements in the given list together using the string as the delimiter. e.g. '---'.join(['aaa', 'bbb', 'ccc']) -> aaa---bbb---ccc

Self-Study

UNICODE & ENCODING

Sequence Types
Unicode Strings

- In 1968, the **A**merican **S**tandard **C**ode for Information Interchange, better known by its acronym **ASCII**, was standardized. ASCII defined numeric codes for various characters, with the numeric values running from 0 to 127. For example, the lowercase letter ‘a’ is assigned 97 as its code value.

- ASCII only defined unaccented characters. There was an ‘**e**’, but no ‘**é**’ or ‘**Í**’. This meant that languages which required accented characters couldn’t be faithfully represented in ASCII.

- In the 1980s, almost all personal computers were **8-bit**, meaning that bytes could hold values ranging from **0** to **255**. Some machines assigned values between **128** and **255** to accented characters.`

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	*[NULL]*	32	20	*[SPACE]*	64	40	**@**	96	60	`
1	1	*[START OF HEADING]*	33	21	**!**	65	41	**A**	97	61	**a**
2	2	*[START OF TEXT]*	34	22	**"**	66	42	**B**	98	62	**b**
3	3	*[END OF TEXT]*	35	23	**#**	67	43	**C**	99	63	**c**
4	4	*[END OF TRANSMISSION]*	36	24	**$**	68	44	**D**	100	64	**d**
5	5	*[ENQUIRY]*	37	25	**%**	69	45	**E**	101	65	**e**
6	6	*[ACKNOWLEDGE]*	38	26	**&**	70	46	**F**	102	66	**f**
7	7	*[BELL]*	39	27	**'**	71	47	**G**	103	67	**g**
8	8	*[BACKSPACE]*	40	28	**(**	72	48	**H**	104	68	**h**
9	9	*[HORIZONTAL TAB]*	41	29	**)**	73	49	**I**	105	69	**i**
10	A	*[LINE FEED]*	42	2A	*	74	4A	**J**	106	6A	**j**
11	B	*[VERTICAL TAB]*	43	2B	+	75	4B	**K**	107	6B	**k**
12	C	*[FORM FEED]*	44	2C	,	76	4C	**L**	108	6C	**l**
13	D	*[CARRIAGE RETURN]*	45	2D	-	77	4D	**M**	109	6D	**m**
14	E	*[SHIFT OUT]*	46	2E	.	78	4E	**N**	110	6E	**n**
15	F	*[SHIFT IN]*	47	2F	/	79	4F	**O**	111	6F	**o**
16	10	*[DATA LINK ESCAPE]*	48	30	**0**	80	50	**P**	112	70	**p**
17	11	*[DEVICE CONTROL 1]*	49	31	**1**	81	51	**Q**	113	71	**q**
18	12	*[DEVICE CONTROL 2]*	50	32	**2**	82	52	**R**	114	72	**r**
19	13	*[DEVICE CONTROL 3]*	51	33	**3**	83	53	**S**	115	73	**s**
20	14	*[DEVICE CONTROL 4]*	52	34	**4**	84	54	**T**	116	74	**t**
21	15	*[NEGATIVE ACKNOWLEDGE]*	53	35	**5**	85	55	**U**	117	75	**u**
22	16	*[SYNCHRONOUS IDLE]*	54	36	**6**	86	56	**V**	118	76	**v**
23	17	*[ENG OF TRANS. BLOCK]*	55	37	**7**	87	57	**W**	119	77	**w**
24	18	*[CANCEL]*	56	38	**8**	88	58	**X**	120	78	**x**
25	19	*[END OF MEDIUM]*	57	39	**9**	89	59	**Y**	121	79	**y**
26	1A	*[SUBSTITUTE]*	58	3A	:	90	5A	**Z**	122	7A	**z**
27	1B	*[ESCAPE]*	59	3B	;	91	5B	[	123	7B	{
28	1C	*[FILE SEPARATOR]*	60	3C	<	92	5C	\	124	7C	|
29	1D	*[GROUP SEPARATOR]*	61	3D	=	93	5D	]	125	7D	}
30	1E	*[RECORD SEPARATOR]*	62	3E	>	94	5E	^	126	7E	~
31	1F	*[UNIT SEPARATOR]*	63	3F	?	95	5F	_	127	7F	*[DEL]*

Sequence Types
Unicode Strings

- **Unicode** started out using **16-bit** characters instead of 8-bit characters. 16 bits means you have 2^16 = **65,536** distinct values available, making it possible to represent many different characters from many different alphabets.
- Modern Unicode specification uses a wider range of codes, **0x10ffff** in base-16 (=**1114111** code).
- The Unicode standard describes how characters are represented by **code points**. A code point is an integer value, usually denoted in base 16. In the standard, a code point is written using the notation **U+12ca** to mean the character with value 0x12ca (4810 decimal).
- Unicode table:
  - http://unicode-table.com/en/

Sequence Types
Unicode Strings

- Unicode strings are expressed as instances of the **unicode** type.

```
unicode('abcdef')   #u'abcdef'
s = unicode('abcdef')
type(s)  #<type 'unicode'>
```

- Unicode literals are written as strings prefixed with the ‘**u**’ or ‘**U**’ character: **u**'**abcdefghijk**'.
- Specific code points can be written using the **\u** escape sequence, which is followed by four hex digits giving the code point, **\U** escape sequence is similar, but expects 8 hex digits, not 4. The **\x** only takes two hex digits.

```
ustring = u'A unicode \u018e string \xf1'
ustring  #u'A unicode \u018e string \xf1'
```

Sequence Types
Unicode Strings

- The built-in **print** does not work fully with unicode strings. You can encode() first to print in utf-8.

```
>>> ustring=u"\u00c0" #code point for ‘À’
>>> print(ustring)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Python27\lib\encodings\cp720.py", line 14, in encode
    return codecs.charmap_encode(input,errors,encoding_table)
UnicodeEncodeError: 'charmap' codec can't encode character u'\xc0'
in position 0
: character maps to <undefined>
```

Unicode & Encoding

- To summarize the previous section:
  - Unicode string is a sequence of code points, which are numbers from 0 to 0x10ffff.
  - This sequence needs to be represented as a set of bytes (meaning, values from 0-255) in memory. The rules for translating a Unicode string into a sequence of bytes are called an **encoding**.
  - Here is a list of encodings that python support:
    - https://docs.python.org/2.7/library/codecs.html#standard-encodings

Unicode & Encoding

- The first encoding you might think of is an **array** of **32-bit** integers. In this representation, the string "**Python**" would look like this:

```
   P           y           t           h           o           n
0x50 00 00 00 79 00 00 00 74 00 00 00 68 00 00 00 6f 00 00 00 6e 00 00 00
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
```

- This representation is straightforward but using it presents a number of problems :
  - It's not portable; different processors order the bytes differently.
  - It's very wasteful of space. In most texts, the majority of the code points are less than 127, or less than 255. The above string takes 24 bytes compared to the 6 bytes needed for an ASCII representation.

Unicode & Encoding

- Encodings don't have to handle every possible Unicode character, and most encodings don't. For example, Python's **default** encoding is the '**ascii**' encoding. The rules for converting a Unicode string into the ASCII encoding are simple; for each code point:
  - If the code point is < 128, each byte is the same as the value of the code point.
  - If the code point is 128 or greater, the Unicode string can't be represented in this encoding. (Python raises a **UnicodeEncodeError** exception in this case.)

- **UTF-8** is one of the most commonly used encodings. UTF stands for "**U**nicode **T**ransformation **F**ormat", and the '**8**' means that **8-bit** numbers are used in the encoding. UTF-8 uses the following rules:
  - If the code point is <128, it's represented by the corresponding byte value.
  - If the code point is between 128 and 0x7ff, it's turned into two byte values between 128 and 255.
  - Code points >0x7ff are turned into three- or four-byte sequences, where each byte of the sequence is between 128 and 255.

Unicode & Encoding

- To convert a unicode string to bytes with an encoding such as 'utf-8', call the **encode**('utf-8') method on the unicode string. Going the other direction, the **unicode**(s, 'utf-8') function converts encoded plain bytes to a unicode string.

```
ustring = u'A unicode \u018e string \xf1'
s = ustring.encode('utf-8')
s  #'A unicode \xc6\x8e string \xc3\xb1'  # bytes of utf-8 encoding
t = unicode(s, 'utf-8')    # Convert bytes back to a unicode string
t == ustring       # It's the same as the original, yay!
```

Unicode & Encoding

- For more information about Unicode and Encoding:
  - https://docs.python.org/2.7/howto/unicode.html?highlight=unicode

Self-Study

UNICODE & ENCODING

Conditional Statements
If …. Elif …. Else

- Python does not use { } to enclose blocks of code for if/loops/function etc.. Instead, Python uses the colon (**:**) and indentation/whitespace to group statements.

- Python has the usual comparison operations: **==**, **!=**, **<**, **<=**, **>**, **>=**, **<>**.

Conditional Statements If …. Elif …. Else

- E.x. A policeman pulling over a speeder -- notice how each block of then/else statements starts with a **:** and the statements are grouped by their **indentation**:

```
if speed >= 80:
  print 'License and registration please'
  if mood == 'terrible' or speed >= 100:
    print 'You have the right to remain silent.'
  elif mood == 'bad' or speed >= 90:
    print "I'm going to have to write you a ticket."
    write_ticket()
  else:
    print "Let's try to keep it under 80 ok?"
```

Conditional Statements
If …. Elif …. Else

- If there will be an empty body for a condition it must be filled with the keyword **None**:
  - **Not Correct**:

```
if institute != 'ITI':
  #do nothing
else:
  print "Long Live ITI"
```

  - **Correct**:

```
if institute != 'ITI':
  None
else:
  print "Long Live ITI"
```

Truth Value

- Any object can be tested for truth value, for use in an **if** or **while** condition or as operand of the **Boolean operations** (and, or, not **instead of** &&, ||, !)
- The following values are considered False as a truthiness value:
  - None
  - False
  - zero of any numeric type, for example, 0, 0L, 0.0, 0j.
  - any empty sequence, for example, '', (), [].
  - any empty mapping, for example, {}.
- All other values are considered True as a truthiness value.

Truth Value

- Example on False truth value:

```
def foo():
    x=None
    if x:
        print 'True'
    else:
        print 'False'   #prints False
```

```
def foo():
    x=[ ]
    if x:
        print 'True'
    else:
        print 'False'   #prints False
```

- Example on True truth value:

```
def foo():
    x=6
    if x:
        print 'True'
    else:
        print 'False'   #prints True
```

```
def foo():
    x=(1,2,3)
    if x:
        print 'True'
    else:
        print 'False'   #prints True
```

Truth Value

- Example on False truth value (Caution):

```
def foo():
    x=None
    if x == False:
        print 'True'
    else:
        print 'False'   #prints False
```

```
def foo():
    x=[ ]
    if x == False:
        print 'True'
    else:
        print 'False'   #prints False
```

- Example on True truth value (Caution):

```
def foo():
    x=6
    if x == True:
        print 'True'
    else:
        print 'False'   #prints False
```

```
def foo():
    x=(1,2,3)
    if x == True:
        print 'True'
    else:
        print 'False'   #prints False
```

LAB – 3
STRINGS LAB

Strings Lab

- Please download the lab from the following link:
  - https://drive.google.com/drive/folders/0B6Hf8UvSSqTXZDRHSVJuLVpkVmM?usp=sharing
- Complete the script **string1.py** in **45** mins and send your solution on the following email:
  - Omar.Soliman@imtSchool.com with the following subject :
    - If you are from ITI-Smart track ES:
      - [ITI-SV-39] [PY-string1] yourfullname
    - If you are from ITI-Nasr City track ES:
      - [ITI-NC-39] [PY-string1] yourfullname
- Complete the script **string2.py** and send it after **2** days max from the session and send it.
  - Use the same subject header as above but replace [PY-string1] with [PY-string2]

LAB – 3
STRINGS LAB

What's Next ?

- Get Certified With:
  - https://www.edx.org/course/learn-program-using-python-utarlingtonx-cse1309x
  - https://www.coursera.org/course/interactivepython1
  - https://www.coursera.org/course/interactivepython2

- More Interesting References:
  - Python Cookbook, 2nd Edition
  - https://automatetheboringstuff.com/
  - http://code.activestate.com/recipes/langs/

Eng. Mohammad A.Hekal: embeddedgeek.34@gmail.com
Eng. Omar Soliman: Omar.Soliman@imtSchool.com