Iceland Liechtenstein Norway grants

Working together for a green, competitive and inclusive Europe

EMERALD

The Education, Scholarships, Apprenticeships and Youth Entrepreneurship

EUROPEAN NETWORK FOR 3D PRINTING OF BIOMIMETIC MECHATRONIC SYSTEMS

MODULE 3 – Computer Programming

Project Title	**European network for 3D printing of biomimetic mechatronic systems 21-COP-0019**
Output	**IO1 - EMERALD e-book for developing of biomimetic mechatronic systems**
Module	**Module 3 – Computer Programming**
Date of Delivery	**July 2022**
Authors	**Martin Zelenay, Michal Gallia**
Version	**Final Version**

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA ROMANIA

UNIVERSITY POLITEHNICA OF BUCHAREST 1818

POLITECHNIKA POZNAŃSKA POZNAN UNIVERSITY OF TECHNOLOGY

UiA University of Agder

bizzcom

Contents

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Introduction

Welcome to Computer Programming Python Class -- this is a free class for people with a little bit of programming experience who want to learn Python. The class includes written materials and lots of code exercises to practice Python coding. These materials are used within to introduce Python to people who have just a little programming experience. The first exercises work on basic Python concepts like strings and lists, building up to the later exercises which are full programs dealing with text files, processes, and http connections. The class is geared for people who have a little bit of programming experience in some language, enough to know what a "variable" or "if statement" is. Beyond that, you do not need to be an expert programmer to use this material.

To get started, the Python sections are Python Set Up to get Python installed on your machine, Python Introduction for an introduction to the language, and then Python Strings starts the coding material, leading to the first exercise. The end of each written section includes a link to the code exercise for that section's material. The lecture videos parallel the written materials, introducing Python, then strings, then first exercises, and so on. All this material makes up an intensive class.

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Python Set Up

Basic definitions

This page explains how to set up Python on a machine so you can run and edit Python programs, and links to the exercise code to download. You can do this before starting the class, or you can leave it until you've gotten far enough in the class that you want to write some code. The Class uses a simple, standard Python installation, although more complex strategies are possible. Python is free and open source, available for all operating systems from python.org. In particular we want a Python install where you can do two things:

- Run an existing python program, such as hello.py
- Run the Python interpreter interactively, so you can type code right at it

Both of the above are done quite a lot in the lecture videos, and it's definitely something you need to be able to do to solve the exercises.

Download Google Python Exercises

As a first step, download the google-python-exercises.zip file and unzip it someplace where you can work on it. The resulting google-python-exercises directory contains many different python code exercises you can work on. In particular, google-python-exercises contains a simple hello.py file you can use in the next step to check that Python is working on your machine. Below are instructions for Windows and other operating systems.

Python on Linux, Mac OS X, and other OS

Most operating systems other than Windows already have Python installed by default. To check that Python is installed, open a command line (typically by running the "Terminal"

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

program), and cd to the google-python-exercises directory. Try the following to run the hello.py program (what you type is shown in bold):

```
~/google-python-exercises$ python hello.py
Hello World
~/google-python-exercises$ python hello.py Alice
Hello Alice
```

If python is not installed, see the Python.org download page. To run the Python interpreter interactively, just type `python` in the terminal:

```
~/google-python-exercises$ python3
Python 3.X.X (XXX, XXX XX XXXX, 03:41:42) [XXX] on XXX
Type "help", "copyright", "credits" or "license" for more information.
>>> 1 + 1
2
>>> you can type expressions here .. use ctrl-d to exit
```

The two lines python prints after you type python and before the >>> prompt tells you about the version of python you're using and where it was built. As long as the first thing printed is "Python 3.", these examples should work for you. This course is designed for Python 3.X or later.

Python on Windows

To install Python on Windows, go to the python.org download page and download Python 3.X.X. Run the Python installer and accept all the defaults. This will install Python in the root directory and set up some file associations.

With Python installed, open a command prompt (Accessories > Command Prompt, or type `cmd` into the run dialog). Cd to the google-python-exercises directory (from unzipping google-

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

python-exercises.zip). You should be able to run the hello.py python program by typing `python hello.py` (what you type is shown in bold):

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

If this works, Python is installed. Otherwise, see Python Windows FAQ for help.

To run the Python interpreter interactively, select the Run... command from the Start menu, and type `Python` -- this will launch Python interactively in its own window. On Windows, use **Ctrl-Z** to exit (on all other operating systems it's **Ctrl-D** to exit).

In the lecture videos, we generally run the Python programs with commands like `./hello.py`. On Windows, it's simplest to use the `python hello.py` form.

Editing Python (all operating systems)

A Python program is just a text file that you edit directly. As above, you should have a command line open, where you can type `python hello.py Alice` to run whatever exercise you are working on. At the command line prompt, just hit the up-arrow key to recall previously typed commands, so it's easy to run previous commands without retyping them.

You want a text editor with a little understanding of code and indentation. There are many good free ones:

Windows -- **do not use Notepad or Wordpad**. Try the free and open source Notepad++ or the free and open source JEdit

Mac -- The built in TextEdit works, but not very well. Try the free BBEdit or the free and open source JEdit

Linux -- any unix text editor is fine, or try the above JEdit.

Editor Settings

Following are some recommended settings for your text editor:

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

When you press **Tab**, it's best if your editor inserts spaces instead of a real tab character. All of the tutorial files use 2-spaces as the indent, and 4-spaces is another popular choice.

It's helpful if the editor will "auto indent" so when you press **Enter**, the new line starts with the same indentation as the previous line.

When you save your files, use the unix line-ending convention, since that's how the various starter files are set up. If running hello.py gives the error "Unknown option: -", the file may have the wrong line-ending.

Here are the preferences to set for common editors to treat tabs and line-endings correctly for Python:

Windows Notepad++ -- Tabs: Settings > Preferences > Edit Components > Tab settings, and Settings > Preferences > MISC for auto-indent. Line endings: Format > Convert, set to Unix.

JEdit (any OS) -- Line endings: Little 'U' 'W' 'M' on status bar, set it to 'U' (for Unix line-endings).

Windows Notepad or Wordpad -- do not use.

Mac BBEdit -- Tabs: At the top, BBEdit > Preferences (or Cmd + , shortcut). Go to Editor Defaults section and make sure Auto-indent and Auto-expand tabs are checked. Line endings: In Preferences go to Text Files section and make sure Unix (LF) is selected under Line breaks.

Mac TextEdit -- do not use.

Unix pico -- Tabs: Esc-q toggles tab mode, Esc-i to turns on auto-indent mode.

Unix emacs -- Tabs: manually set tabs-inserts-spaces mode: `M-x set-variable(return) indent-tabs-mode(return) nil`.

Editing Check

To try out your editor, edit the hello.py program. Change the word "Hello" in the code to the word "Howdy" (you don't need to understand all the other Python code in there - we'll explain it all in class). Save your edits and run the program to see its new output. Try adding a `print 'yay!'` just below the existing print and with the same indentation. Try running the

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

program, to see that your edits work correctly. For class we want an edit/run workflow that allows you to switch between editing and running easily.

Quick Python Style

One of the advantages of Python is that it makes it easy to type a little code and quickly see what it does. In class, we want a work setup that matches that: a text editor working on the current file.py, and a separate command line window where you can just press the up-arrow key to run file.py and see what it does.

Teaching philosophy aside: the interpreter is great for little experiments, as shown throughout the lectures. However, the exercises are structured as Python files that students edit. Since being able to write Python programs is the ultimate goal, it's best to be in that mode the whole time and use the interpreter just for little experiments.

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Python Introduction

Language Introduction

Python is a dynamic, interpreted (bytecode-compiled) language. There are no type declarations of variables, parameters, functions, or methods in source code. This makes the code short and flexible, and you lose the compile-time type checking of the source code. Python tracks the types of all values at runtime and flags code that does not make sense as it runs.

An excellent way to see how Python code works is to run the Python interpreter and type code right into it. If you ever have a question like, "What happens if I add an `int` to a `list`?" Just typing it into the Python interpreter is a fast and likely the best way to see what happens. (See below to see what really happens!)

```
   $ python         ## Run the Python interpreter
Python 3.X.X (XXX, XXX XX XXXX, 03:41:42) [XXX] on XXX
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: can only concatenate str (not "int") to str
>>> 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
```

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
NameError: name 'foo' is not defined
>>> ^D          ## type CTRL-d to exit (CTRL-z in Windows/DOS terminal)
```

The two lines python prints after you type python and before the >>> prompt tells you about the version of python you're using and where it was built. As long as the first thing printed is "Python 3.", these examples should work for you.

As you can see above, it's easy to experiment with variables and operators. Also, the interpreter throws, or "raises" in Python parlance, a runtime error if the code tries to read a variable that has not been assigned a value. Like C++ and Java, Python is case sensitive so "a" and "A" are different variables. The end of a line marks the end of a statement, so unlike C++ and Java, Python does not require a semicolon at the end of each statement. Comments begin with a '#' and extend to the end of the line.

Python source code

Python source files use the ".py" extension and are called "modules." With a Python module `hello.py`, the easiest way to run it is with the shell command "python hello.py Alice" which calls the Python interpreter to execute the code in `hello.py`, passing it the command line argument "Alice". See the official docs page on all the different options you have when running Python from the command-line.

Here's a very simple `hello.py` program (notice that blocks of code are delimited strictly using indentation rather than curly braces — more on this later!):

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

# 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.
```

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
if__name__ == '__main__':
    main()
```

Running this program from the command line looks like:

```
$ python hello.py Guido
Hello there Guido
$ ./hello.py Alice  ## without needing 'python' first (Unix)
Hello there Alice
```

Imports, Command-line arguments, and ***`len()`***

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. A Python module can be run directly — as above "python hello.py Bob" — 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__==...` shown above to call a main() function when the module is run directly, but not when the module is imported by some other module.

In a standard Python program, the list `sys.argv` contains the command-line arguments in the standard way with sys.argv[0] being the program itself, sys.argv[1] the first argument, and so on. If you know about `argc`, or the number of arguments, you can simply request this value from Python with `len(sys.argv)`, just like we did in the interactive interpreter code above when requesting the length of a string. In general, `len()` can tell you how long a string is, the number of elements in lists and tuples (another array-like data structure), and the number of key-value pairs in a dictionary.

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.
    """
```

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
    result = s + s + s # can also use "s * 3" which is faster (Why?)
    if exclaim:
        result = result + '!!!!'
    return result
```

Notice also how the lines that make up the function or if-statement are grouped by all having the same level of indentation. We also presented 2 different ways to repeat strings, using the + operator which is more user-friendly, but * also works because it's Python's "repeat" operator, meaning that `'-' * 10` gives `'----------'`, a neat way to create an onscreen "line." In the code comment, we hinted that * works faster than +, the reason being that * calculates the size of the resulting object once whereas with +, that calculation is made each time + is called. Both + and * are called "overloaded" operators because they mean different things for numbers vs. for strings (and other data types).

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. The docstring can be a single line, or a multi-line description as in the example above. (Yes, those are "triple quotes," a feature unique to Python!) Variables defined in the function are local to that function, so the "result" in the above 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!!!
```

At run time, functions must be defined by the execution of a "def" before they are called. It's typical to def a main() function towards the bottom of the file with the functions it calls above it.

Indentation

One unusual Python feature is that the whitespace indentation of a piece of code affects its meaning. A logical block of statements such as the ones that make up a function should all have the same indentation, set in from the indentation of their parent function or "if" or

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

whatever. If one of the lines in a group has a different indentation, it is flagged as a syntax error.

Python's use of whitespace feels a little strange at first, but it's logical and I found I got used to it very quickly. Avoid using TABs as they greatly complicate the indentation scheme (not to mention TABs may mean different things on different platforms). Set your editor to insert spaces instead of TABs for Python code.

A common question beginners ask is, "How many spaces should I indent?" According to the official Python style guide (PEP 8), you should indent with 4 spaces. (Fun fact: Google's internal style guideline dictates indenting by 2 spaces!)

Code Checked at Runtime

Python does very little checking at compile time, deferring almost all type, name, etc. checks on each line until that line runs. Suppose the above main() calls repeat() like this:

```
def main():
 if name == 'Guido':
     print(repeeeet(name) + '!!!')
 else:
     print(repeat(name))
```

The if-statement contains an obvious error, where the repeat() function is accidentally typed in as repeeeet(). The funny thing in Python ... this code compiles and runs fine so long as the name at runtime is not 'Guido'. Only when a run actually tries to execute the repeeeet() will it notice that there is no such function and raise an error. There is also a second error in this snippet. name wasn't assigned a value before it is compared with 'Guido'. Python will raise a 'NameError' if you try to evaluate an unassigned variable. These are some examples demonstrating that when you first run a Python program, some of the first errors you see will be simple typos or uninitialized variables like these. This is one area where languages with a more verbose type system, like Java, have an advantage ... they can catch such errors at compile time (but of course you have to maintain all that type information ... it's a tradeoff).

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Variable Names

Since Python variables don't have any type spelled out in the source code, it's extra helpful to give meaningful names to your variables to remind yourself of what's going on. So use "name" if it's a single name, and "names" if it's a list of names, and "tuples" if it's a list of tuples. Many basic Python errors result from forgetting what type of value is in each variable, so use your variable names (all you have really) to help keep things straight.

As far as actual naming goes, some languages prefer underscored_parts for variable names made up of "more than one word," but other languages prefer camelCasing. In general, Python prefers the underscore method but guides developers to defer to camelCasing if integrating into existing Python code that already uses that style. Readability counts. Read more in the section on naming conventions in PEP 8.

As you can guess, keywords like 'if' and 'while' cannot be used as variable names — you'll get a syntax error if you do. However, be careful not to use built-ins as variable names. For example, while 'str', 'list' and 'print' may seem like good names, you'd be overriding those system variables. Built-ins are not keywords and thus, are susceptible to inadvertent use by new Python developers.

More on Modules and their Namespaces

Suppose you've got a module "binky.py" which contains a "def foo()". The fully qualified name of that foo function is "binky.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 binky, module1, and module2 each have their own "namespaces," which as you can guess are variable name-to-object bindings.

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 make them available by their fully-qualified name, e.g. sys.exit(). (Yes, 'sys' has a namespace too!)

```
import sys
```

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

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

There is another import form that looks like this: "from sys import argv, exit". That makes argv and exit() available by their short names; however, we recommend the original form with the fully-qualified names because it's a lot easier to determine where a function or attribute came from.

There are many modules and packages which are bundled with a standard installation of the Python interpreter, so you don't have to do anything extra to use them. These are collectively known as the "Python Standard Library." Commonly used modules/packages include:

sys — access to exit(), argv, stdin, stdout, ...

re — regular expressions

os — operating system interface, file system

You can find the documentation of all the Standard Library modules and packages at http://docs.python.org/library.

Online help, ***`help()`****, and* ***`dir()`***

There are a variety of ways to get help for Python.

Do a Google search, starting with the word "python", like "python list" or "python string lowercase". The first hit is often the answer. This technique seems to work better for Python than it does for other languages for some reason.

The official Python docs site — docs.python.org — has high quality docs. Nonetheless, I often find a Google search of a couple words to be quicker.

There is also an official Tutor mailing list specifically designed for those who are new to Python and/or programming!

Many questions (and answers) can be found on StackOverflow and Quora.

Use the help() and dir() functions (see below).

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Inside the Python interpreter, the help() function pulls up documentation strings for various modules, functions, and methods. These doc strings are similar to Java's javadoc. The dir() function tells you what the attributes of an object are. Below are some ways to call help() and dir() from the interpreter:

`help(len)` — help string for the built-in `len()` function; note that it's "len" not "len()", which is a **call** to the function, which we don't want

`help(sys)` — help string for the `sys` module (must do an `import sys` first)

`dir(sys)` — `dir()` is like `help()` but just gives a quick list of its defined symbols, or "attributes"

`help(sys.exit)` — help string for the `exit()` function in the `sys` module

`help('xyz'.split)` — help string for the `split()` method for string objects. You can call `help()` with that object itself or an **example** of that object, plus its attribute. For example, calling `help('xyz'.split)` is the same as calling `help(str.split)`.

`help(list)` — help string for `list` objects

`dir(list)` — displays `list` object attributes, including its methods

`help(list.append)` — help string for the `append()` method for `list` objects

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Python Strings

Python has a built-in string class named "str" with many handy features (there is an older module named "string" which you should not use). String literals can be enclosed by either double or single quotes, although single quotes are more commonly used. Backslash escapes work the usual way within both single and double quoted literals -- e.g. \n \' \". A double quoted string literal can contain single quotes without any fuss (e.g. "I didn't do it") and likewise single quoted string can contain double quotes. A string literal can span multiple lines, but there must be a backslash \ at the end of each line to escape the newline. String literals inside triple quotes, """ or ''', can span multiple lines of text.

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'.

Characters in a string can be accessed using the standard [ ] syntax, and like Java and C++, Python uses zero-based indexing, so if s is 'hello' s[1] is 'e'. If the index is out of bounds for the string, Python raises an error. The Python style (unlike Perl) is to halt if it can't tell what to do, rather than just make up a default value. The handy "slice" syntax (below) also works to extract any substring from a string. The len(string) function returns the length of a string. The [ ] syntax and the len() function actually work on any sequence type -- strings, lists, etc.. Python tries to make its operations work consistently across different types. Python newbie gotcha: don't use "len" as a variable name to avoid blocking out the len() function. The '+' operator can concatenate two strings. Notice in the code below that variables are not pre-declared -- just assign to them and go.

```
    s = 'hi'
print s[1]          ## i
print len(s)        ## 2
print s + ' there'  ## hi there
```

Unlike Java, the '+' does not automatically convert numbers or other types to string form. The str() function converts values to a string form so they can be combined with other strings.

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
    pi = 3.14
##text = 'The value of pi is ' + pi      ## NO, does not work
text = 'The value of pi is ' + str(pi)   ## yes
```

For numbers, the standard operators, +, /, * work in the usual way. There is no ++ operator, but +=, -=, etc. work. If you want integer division, use 2 slashes -- e.g. 6 // 5 is 1

The "print" function normally prints out one or more python items followed by a newline. A "raw" string literal is prefixed by an 'r' and passes all the chars through without special treatment of backslashes, so r'x\nx' evaluates to the length-4 string 'x\nx'. "print" can take several arguments to change how it prints things out (see python.org print function definition) like setting "end" to "" to no longer print a newline after it finishes printing out all of the items.

```
    raw = r'this\t\n and that'

# this\t\n and that
print(raw)

multi = """It was the best of times.
It was the worst of times."""

# It was the best of times.
#   It was the worst of times.
print(multi)
```

String Methods

Here are some of the most common string methods. A method is like a function, but it runs "on" an object. If the variable s is a string, then the code s.lower() runs the lower() method on that string object and returns the result (this idea of a method running on an object is one of the basic ideas that make up Object Oriented Programming, OOP). Here are some of the most common 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

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

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

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

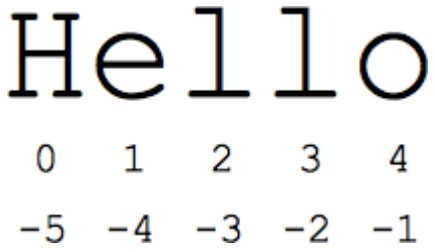
A google search for "python str" should lead you to the official python.org string methods which lists all the str methods.

Python does not have a separate character type. Instead an expression like s[8] returns a string-length-1 containing the character. With that string-length-1, the operators ==, <=, ... all work as you would expect, so mostly you don't need to know that Python does not have a separate scalar "char" type.

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. Suppose we have s = "Hello"

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.



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

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

The standard zero-based index numbers give easy access to chars near the start of the string. As an alternative, Python uses negative numbers to give easy access to the chars at the end of the string: s[-1] is the last char 'o', s[-2] is 'l' the next-to-last char, and so on. Negative index numbers count back from the end of the string:

s[-1] is 'o' -- last char (1st from the end)

s[-4] is 'e' -- 4th 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.

It is a neat truism of slices that for any index n, `s[:n] + s[n:] == s`. This works even for n negative or out of bounds. Or put another way s[:n] and s[n:] always partition the string into two string parts, conserving all the characters. As we'll see in the list section later, slices work with lists too.

String formatting

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

One neat thing python can do is automatically convert objects into a string suitable for printing. Two built-in ways to do this are formatted string literals, also called "f-strings", and invoking str.format().

Formatted string literals

You'll often see formatted string literals used in situations like:

```
    value = 2.791514
print(f'approximate value = {value:.2f}')  # approximate value = 2.79

car = {'tires':4, 'doors':2}
print(f'car = {car}') # car = {'tires': 4, 'doors': 2}
```

A formatted literal string is prefixed with 'f' (like the 'r' prefix used for raw strings). Any text outside of curly braces '{}' is printed out directly. Expressions contained in '{}' are are printed out using the format specification described in the format spec. There are lots of neat things you can do with the formatting including truncation and conversion to scientific notation and left/right/center alignment.

f-strings are very useful when you'd like to print out a table of objects and would like the columns representing different object attributes to be aligned like

```
    address_book = [{'name':'N.X.', 'addr':'15 Jones St', 'bonus': 70},
     {'name':'J.P.', 'addr':'1005 5th St', 'bonus': 400},
     {'name':'A.A.', 'addr':'200001 Bdwy', 'bonus': 5},]

for person in  address_book:
  print(f'{person["name"]:8} || {person["addr"]:20} ||
{person["bonus"]:>5}')

# N.X.     || 15 Jones St          ||    70
# J.P.     || 1005 5th St          ||   400
# A.A.     || 200001 Bdwy          ||     5
```

String %

Python also has an older 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 parentheses):

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
    # % operator
  text = "%d little pigs come out, or I'll %s, and I'll %s, and I'll blow
your %s down." % (3, 'huff', 'puff', 'house')
```

The above line is kind of long -- suppose you want to break it into separate lines. You cannot just split the line after the '%' as you might in other languages, since by default Python treats each line as a separate statement (on the plus side, this is why we don't need to type semi-colons on each line). To fix this, enclose the whole expression in an outer set of parenthesis -- then the expression is allowed to span multiple lines. This code-across-lines technique works with the various grouping constructs detailed below: ( ), [ ], { }.

```
    # Add parentheses to make the long line work:
  text = (
    "%d little pigs come out, or I'll %s, and I'll %s, and I'll blow your
%s down."
    % (3, 'huff', 'puff', 'house'))
```

That's better, but the line is still a little long. Python lets you cut a line up into chunks, which it will then automatically concatenate. So, to make this line even shorter, we can do this:

```
    # Split the line into chunks, which are concatenated automatically by
Python
  text = (
    "%d little pigs come out, "
    "or I'll %s, and I'll %s, "
    "and I'll blow your %s down."
    % (3, 'huff', 'puff', 'house'))
```

Strings (Unicode vs bytes)

Regular Python strings are unicode.

Python also supports strings composed of plain bytes (denoted by the prefix 'b' in front of a string literal) like:

```
  > byte_string = b'A byte string'
> byte_string
  b'A byte string'
```

A unicode string is a different type of object from a byte string but various libraries such as regular expressions work correctly if passed either type of string.

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

To convert a regular Python string to bytes, call the encode() method on the string. Going the other direction, the byte string decode() method converts encoded plain bytes to a unicode string:

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

In the file-reading section, there's an example that shows how to open a text file with some encoding and read out unicode strings.

If Statement

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. The boolean test for an if does not need to be in parenthesis (big difference from C++/Java), and it can have *elif* and *else* clauses (mnemonic: the word "elif" is the same length as the word "else").

Any value can be used as an if-test. The "zero" values all count as false: None, 0, empty string, empty list, empty dictionary. There is also a Boolean type with two values: True and False (converted to an int, these are 1 and 0). Python has the usual comparison operations: ==, !=, <, <=, >, >=. Unlike Java and C, == is overloaded to work correctly with strings. The boolean operators are the spelled out words *and*, *or*, *not* (Python does not use the C-style && || !). Here's what the code might look like for a health app providing drink recommendations throughout the day -- notice how each block of then/else statements starts with a : and the statements are grouped by their indentation:

```
 if time_hour >= 0 and time_hour <= 24:
print('Suggesting a drink option...')
if mood == 'sleepy' and time_hour < 10:
```

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
    print('coffee')
  elif mood == 'thirsty' or time_hour < 2:
    print('lemonade')
  else:
    print('water')
```

I find that omitting the ":" is my most common syntax mistake when typing in the above sort of code, probably since that's an additional thing to type vs. my C++/Java habits. Also, don't put the boolean test in parentheses -- that's a C/Java habit. If the code is short, you can put the code on the same line after ":", like this (this applies to functions, loops, etc. also), although some people feel it's more readable to space things out on separate lines.

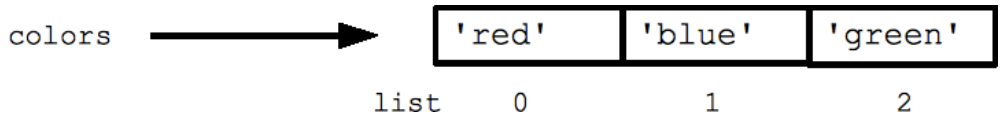
```
   if time_hour < 10: print('coffee')
else: print('water')
```

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Python Lists

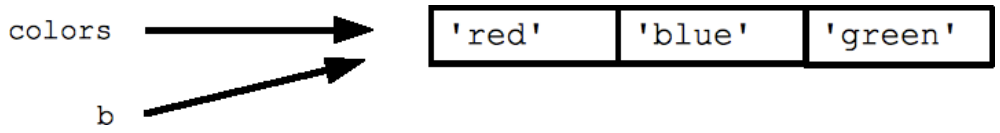
Python has a great built-in list type named "list". List literals are written within square brackets [ ]. Lists work similarly to strings -- use the len() function and square brackets [ ] to access data, with the first element at index 0. (See the official python.org list docs.)

```
    colors = ['red', 'blue', 'green']
print(colors[0])    ## red
print(colors[2])    ## green
print(len(colors))  ## 3
```



Assignment with an = on lists does not make a copy. Instead, assignment makes the two variables point to the one list in memory.

```
    b = colors   ## Does not copy the list
```



The "empty list" is just an empty pair of brackets [ ]. The '+' works to append two lists, so [1, 2] + [3, 4] yields [1, 2, 3, 4] (this is just like + with strings).

FOR and IN

Python's *for* and *in* constructs are extremely useful, and the first use of them we'll see is with lists. The *for* construct -- `for var in list` -- is an easy way to look at each element in a list (or other collection). Do not add or remove from the list during iteration.

```
    squares = [1, 4, 9, 16]
sum = 0
for num in squares:
```

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
    sum += num
print(sum)   ## 30
```

If you know what sort of thing is in the list, use a variable name in the loop that captures that information such as "num", or "name", or "url". Since Python code does not have other syntax to remind you of types, your variable names are a key way for you to keep straight what is going on. (This is a little misleading. As you gain more exposure to python, you'll see references to type hints which allow you to add typing information to your function definitions. Python doesn't use these type hints when it runs your programs. They are used by other programs such as IDEs (integrated development environments) and static analysis tools like linters/type checkers to validate if your functions are called with compatible arguments.)

The *in* construct on its own is an easy way to test if an element appears in a list (or other collection) -- **`value`** `in` **`collection`** -- tests if the value is in the collection, returning True/False.

```
    list = ['larry', 'curly', 'moe']
if 'curly' in list:
  print('yay')
```

The for/in constructs are very commonly used in Python code and work on data types other than list, so you should just memorize their syntax. You may have habits from other languages where you start manually iterating over a collection, where in Python you should just use for/in.

You can also use for/in to work on a string. The string acts like a list of its chars, so `for ch in s: print(ch)` prints all the chars in a string.

Range

The range(n) function yields the numbers 0, 1, ... n-1, and range(a, b) returns a, a+1, ... b-1 -- up to but not including the last number. The combination of the for-loop and the range() function allow you to build a traditional numeric for loop:

```
    ## print the numbers from 0 through 99
for i in range(100):
  print(i)
```

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

There is a variant xrange() which avoids the cost of building the whole list for performance sensitive cases (in Python 3, range() will have the good performance behavior and you can forget about xrange()).

While Loop

Python also has the standard while-loop, and the *break* and *continue* statements work as in C++ and Java, altering the course of the innermost loop. The above for/in loops solves the common case of iterating over every element in a list, but the while loop gives you total control over the index numbers. Here's a while loop which accesses every 3rd element in a list:

```
    ## Access every 3rd element in a list
i = 0
while i < len(a):
  print(a[i])
  i = i + 3
```

List Methods

Here are some other common list methods.

- list.append(elem) -- adds a single element to the end of the list. Common error: does not return the new list, just modifies the original.
- list.insert(index, elem) -- inserts the element at the given index, shifting elements to the right.
- list.extend(list2) adds the elements in list2 to the end of the list. Using + or += on a list is similar to using extend().
- list.index(elem) -- searches for the given element from the start of the list and returns its index. Throws a ValueError if the element does not appear (use "in" to check without a ValueError).
- list.remove(elem) -- searches for the first instance of the given element and removes it (throws ValueError if not present)

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

- list.sort() -- sorts the list in place (does not return it). (The sorted() function shown later is preferred.)
- list.reverse() -- reverses the list in place (does not return it)
- list.pop(index) -- removes and returns the element at the given index. Returns the rightmost element if index is omitted (roughly the opposite of append()).

Notice that these are *methods* on a list object, while len() is a function that takes the list (or string or whatever) as an argument.

```
list = ['larry', 'curly', 'moe']
list.append('shemp')         ## append elem at end
list.insert(0, 'xxx')        ## insert elem at index 0
list.extend(['yyy', 'zzz'])  ## add list of elems at end
print(list)  ## ['xxx', 'larry', 'curly', 'moe', 'shemp', 'yyy', 'zzz']
print(list.index('curly'))    ## 2

list.remove('curly')         ## search and remove that element
list.pop(1)                  ## removes and returns 'larry'
print(list)  ## ['xxx', 'moe', 'shemp', 'yyy', 'zzz']
```

Common error: note that the above methods do not *return* the modified list, they just modify the original list.

```
   list = [1, 2, 3]
print(list.append(4))   ## NO, does not work, append() returns None
## Correct pattern:
list.append(4)
print(list)  ## [1, 2, 3, 4]
```

List Build Up

One common pattern is to start a list as the empty list [], then use append() or extend() to add elements to it:

```
   list = []           ## Start as the empty list
list.append('a')   ## Use append() to add elements
list.append('b')
```

List Slices

Slices work on lists just as with strings, and can also be used to change sub-parts of the list.

```
   list = ['a', 'b', 'c', 'd']
print(list[1:-1])   ## ['b', 'c']
```

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
list[0:2] = 'z'     ## replace ['a', 'b'] with ['z']
print(list)          ## ['z', 'c', 'd']
```

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Summary

Python has become one of the most popular programming languages in the world in recent years. It's used in everything from machine learning to building websites and software testing. It can be used by developers and non-developers alike.

Python, one of the most popular programming languages in the world, has created everything from Netflix's recommendation algorithm to the software that controls self-driving cars. Python is a general-purpose language, which means it's designed to be used in a range of applications, including **data science**, **software and web development**, **automation**, and generally getting stuff done.

Let's take a closer look at what Python is, what it can do, and how you can start learning it.

Python is a computer programming language often used to build websites and software, automate tasks, and conduct data analysis. Python is a general-purpose language, meaning it can be used to create a variety of different programs and isn't specialized for any specific

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

problems. This versatility, along with its beginner-friendliness, has made it one of the most-used programming languages today. A survey conducted by industry analyst firm RedMonk found that it was the second-most popular programming language among developers in 2021 [1].

What is Python used for?

Python is commonly used for developing websites and software, task automation, data analysis, and data visualization. Since it's relatively easy to learn, Python has been adopted by many non-programmers such as accountants and scientists, for a variety of everyday tasks, like organizing finances.

"Writing programs is a very creative and rewarding activity," says University of Michigan and Coursera instructor Charles R Severance in his book *Python for Everybody.* "You can write programs for many reasons, ranging from making your living to solving a difficult data analysis problem to having fun to helping someone else solve a problem."

What can you do with python? Some things include:

- Data analysis and machine learning
- Web development
- Automation or scripting
- Software testing and prototyping
- Everyday tasks

Why is Python so popular?

Python is popular for a number of reasons. Here's a deeper look at what makes it so versatile and easy to use for coders.

It has a **simple syntax** that mimics natural language, so it's easier to read and understand. This makes it quicker to build projects, and faster to improve on them.

It's **versatile**. Python can be used for many different tasks, from web development to machine learning.

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

It's **beginner friendly**, making it popular for entry-level coders.

It's **open source**, which means it's free to use and distribute, even for commercial purposes.

Python's archive of **modules and libraries**—bundles of code that third-party users have created to expand Python's capabilities—is vast and growing.

Python has a **large and active community** that contributes to Python's pool of modules and libraries, and acts as a helpful resource for other programmers. The vast support community means that if coders run into a stumbling block, finding a solution is relatively easy; somebody is bound to have encountered the same problem before.

This project has been funded with support from the Iceland Liechtenstein Norway Grants. This publication [communication] reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.