DSC510: Introduction to Data Science and Analytics
Lab 1: Introduction to Python

University of Cyprus
Department of
Computer Science

Pavlos Antoniou

Office: B109, FST01

Why Python?

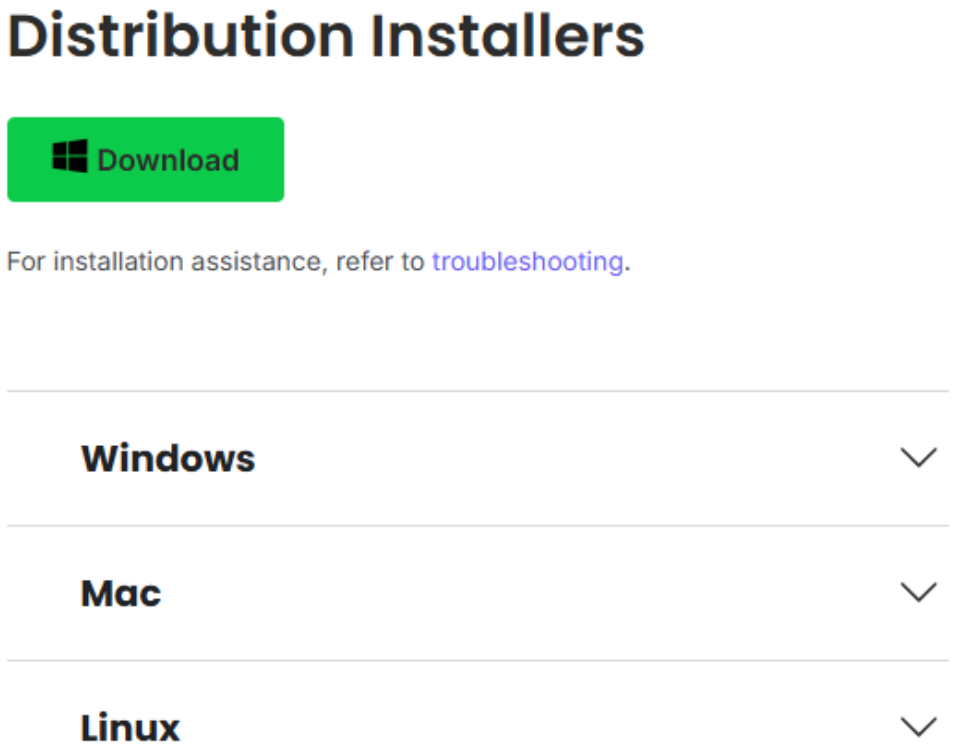
- One of the best languages used by data scientists for various data science projects/applications
- Open source, high level language
- Simple syntax, easy to use
  - Python’s syntax, or the words and symbols used in order to make a program are simple and intuitive. They're basically English words!
  - Easy to learn even without having any programming background
- Provides great libraries to deal with data:
  - import, store, handle, visualize
  - perform statistical analysis
  - train algorithms, build models to cluster/classify or predict quantities

Available Python Distributions

- Official Python website: https://www.python.org/
  - Executables for Windows, macOS, Unix: https://www.python.org/downloads/
  - Includes basic data types and associated functions to operate on them
  - However, … you need to install numerous Data Science packages (numpy, pandas, sklearn) using *pip* (package management system)
- Anaconda Data Science Platform: https://www.anaconda.com/
  - Free for individuals, open-source, easy-to-install Python distribution with over a 1500 packages (including the package management system *conda*) and a GUI named Anaconda Navigator
    - Through Anaconda Navigator you can gain access to some very useful applications such as Jupyter Notebook and Spyder IDE
  - Powered by Anaconda company, but with free community support

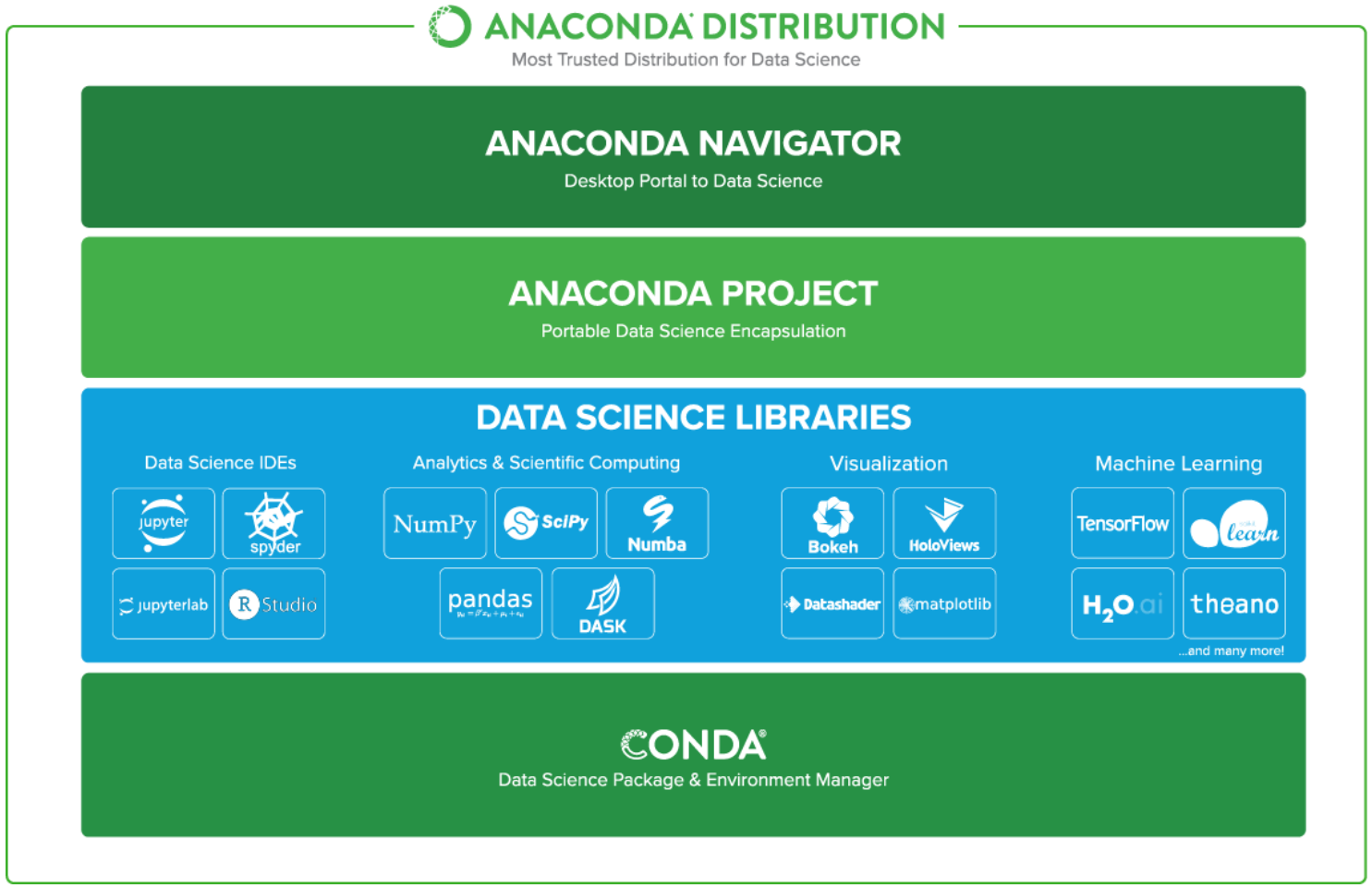
Install Anaconda

- Step 1: Go to https://www.anaconda.com
- Step 2: Download and install the version of Anaconda targeting your Operating System (prefer 64 bit). For all available installers see here: https://www.anaconda.com/download/success



Getting started with Anaconda

- Anaconda Distribution contains conda and Anaconda Navigator as well as Python and hundreds of scientific packages.



Courtesy of https://medium.com/@kumarankita764/new-features-of-anaconda-5-3-5bfdfe9b4240

Getting started with Anaconda

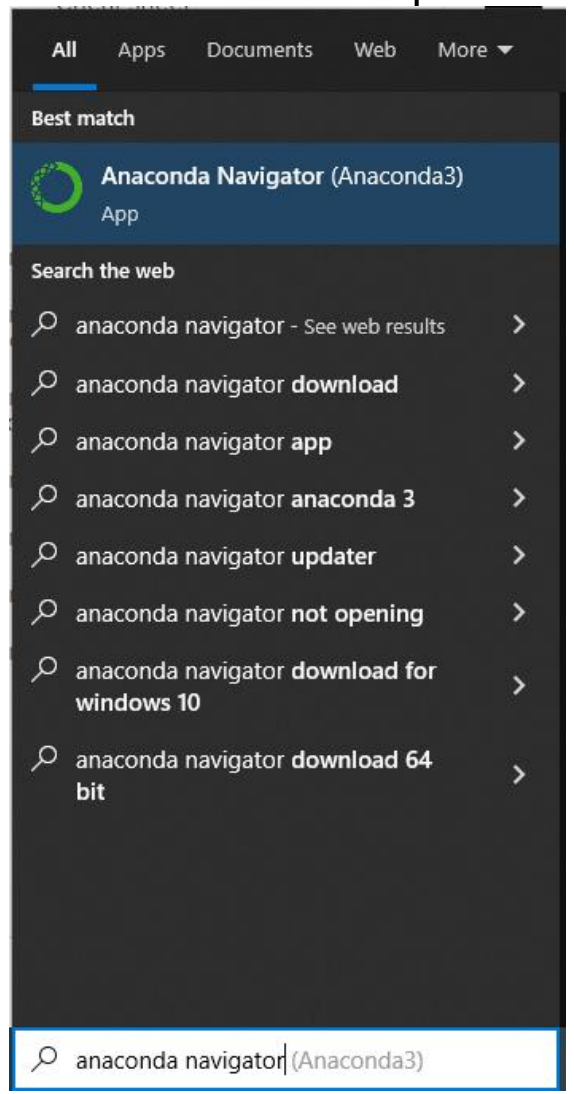
- Conda works as a command line interface called Anaconda Prompt on Windows and via terminal on macOS and Linux.
  - Package, dependency and environment management system
    - quickly installs and updates packages and their dependencies

- Navigator is a desktop graphical user interface (GUI) that allows you to launch applications and easily manage conda packages.

Python program using Anaconda Navigator

- Launch Anaconda Navigator on Windows:
  - Go to Start Menu and type “anaconda navigator” →
- Launch Anaconda Navigator on macOS:
  - Open Launchpad, then click the Anaconda Navigator icon:

- Launch Anaconda Navigator on Unix/Linux:
  - Open a terminal and type “anaconda-navigator”



Anaconda Navigator

File Help

ANACONDA.NAVIGATOR

Connect

Home

Environments

Learning

Community

Applications on base (root) Channels

CMD.exe Prompt
0.1.1
Run a cmd.exe terminal with your current environment from Navigator activated
Launch

Datalore
Online Data Analysis Tool with smart coding assistance by JetBrains. Edit and run your Python notebooks in the cloud and share them with your team.
Launch

IBM Watson Studio Cloud
IBM Watson Studio Cloud provides you the tools to analyze and visualize data, to cleanse and shape data, to create and train machine learning models. Prepare data and build models, using open source data science tools or visual modeling.
Launch

JupyterLab
3.3.2
An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.
Launch

Notebook
6.4.8
Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.
Launch

Powershell Prompt
0.0.1
Run a Powershell terminal with your current environment from Navigator activated
Launch

Qt Console
5.1.1
PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.
Launch

Spyder
5.1.5
Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features
Launch

VS Code
1.66.1
Streamlined code editor with support for development operations like debugging, task running and version control.
Launch

Glueviz
1.0.0
Multidimensional data visualization across files. Explore relationships within and among related datasets.
Install

Orange 3
3.26.0
Component based data mining framework. Data visualization and data analysis for novice and expert. Interactive workflows with a large toolbox.
Install

PyCharm Professional
A full-fledged IDE by JetBrains for both Scientific and Web Python development. Supports HTML, JS, and SQL.
Install

RStudio
1.1.456
A set of integrated tools designed to help you be more productive with R. Includes R essentials and notebooks.

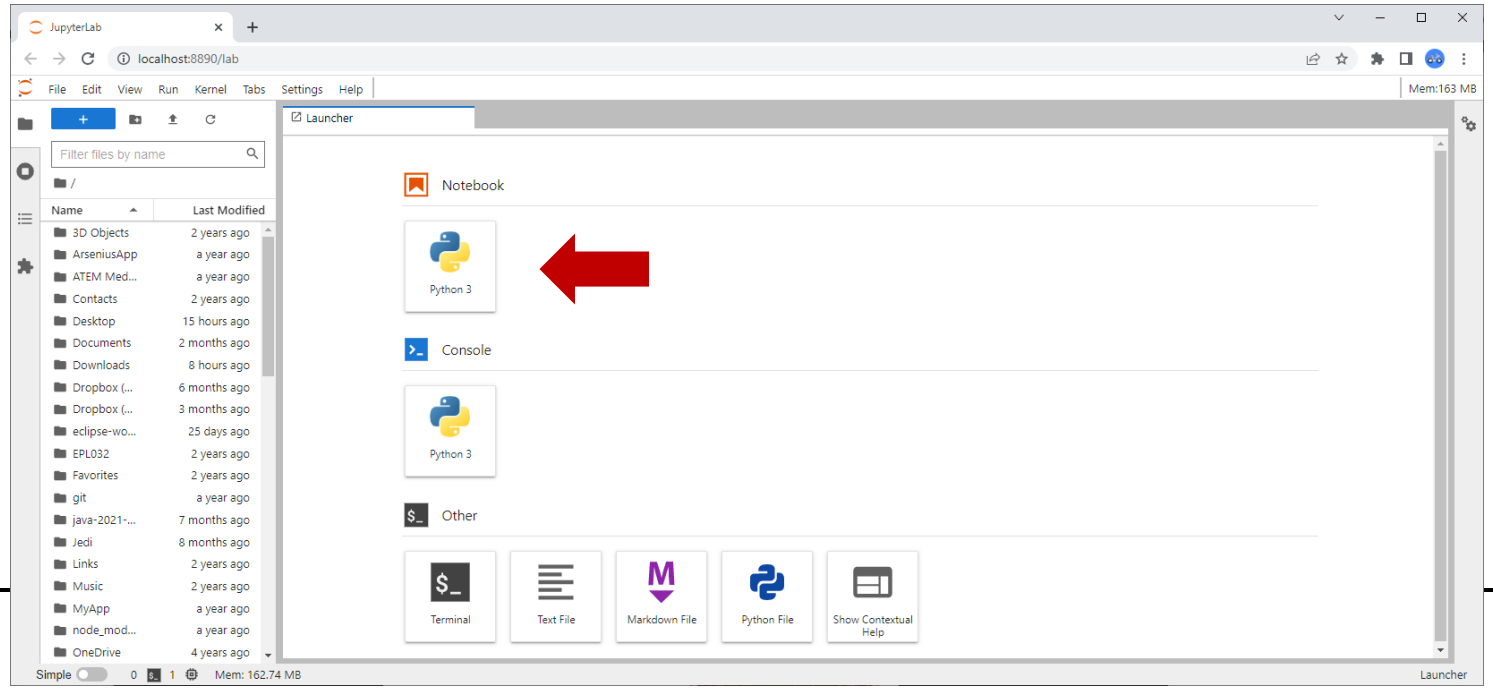
Premium packages and dedicated support.

Documentation

Anaconda Blog

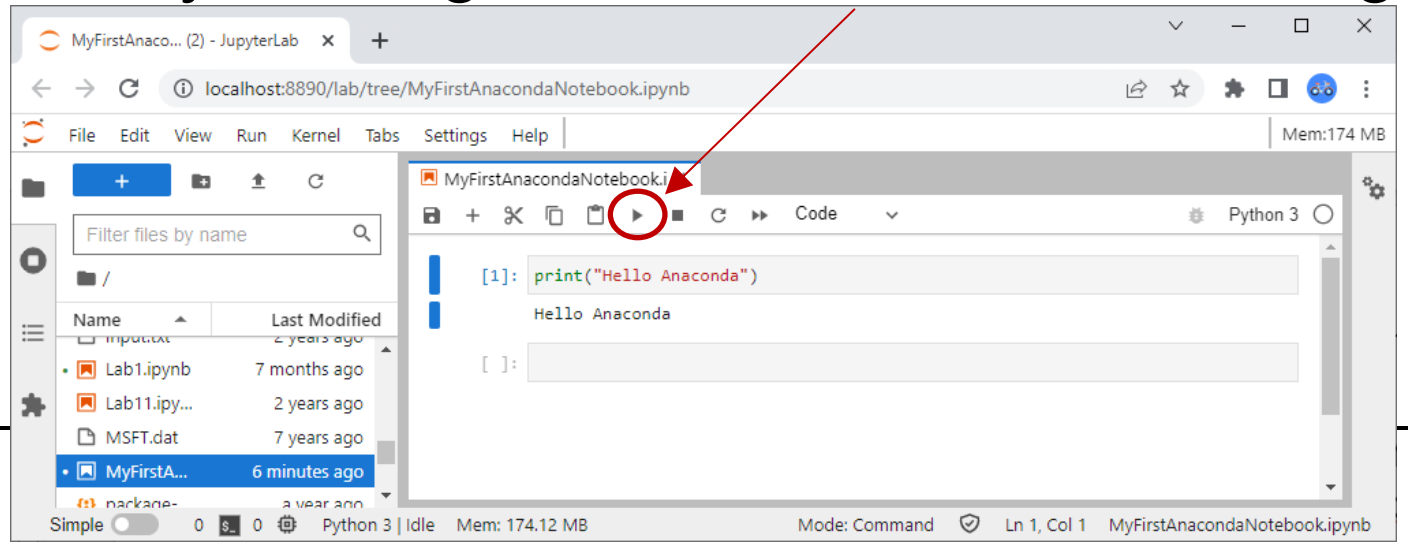
Write/Run Python programs in JupyterLab

- Launch JupyterLab by clicking JupyterLab's Launch button.
- This will launch a new browser window (or a new tab on an existing browser window) showing the Notebook Dashboard.
- In the middle of the Launcher tab there is a button labeled with the Python version you installed. Click it and create a new Notebook.

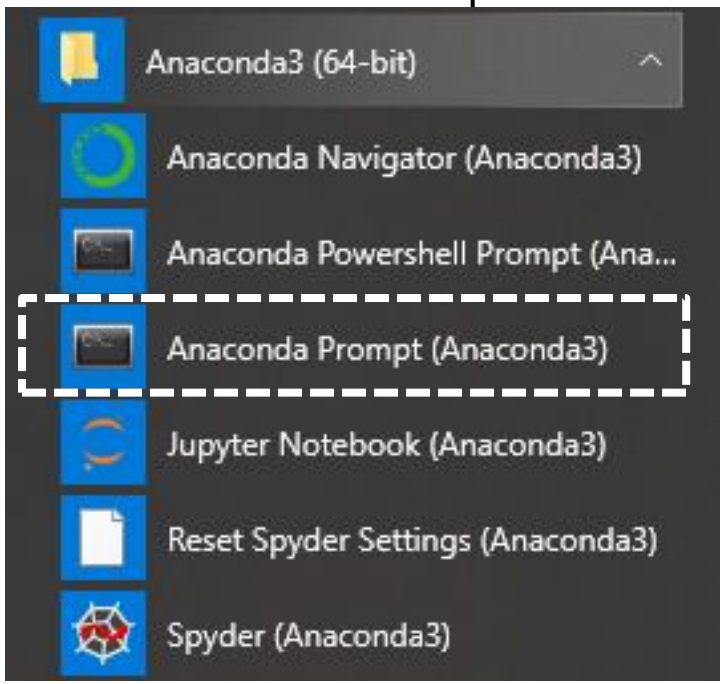


Run Python in JupyterLab

- Rename your Notebook
  - Right click on the filename (shown on the tab). You can rename it to whatever you'd like, but for this example we'll use MyFirstAnacondaNotebook.ipynb
- In the first line of the Notebook, type `print("Hello Anaconda")`
- Save your Notebook by either clicking the save and create checkpoint icon or by selecting File → Save Notebook in top menu.
- Run your new program by clicking the Run button or selecting Run from the top menu.

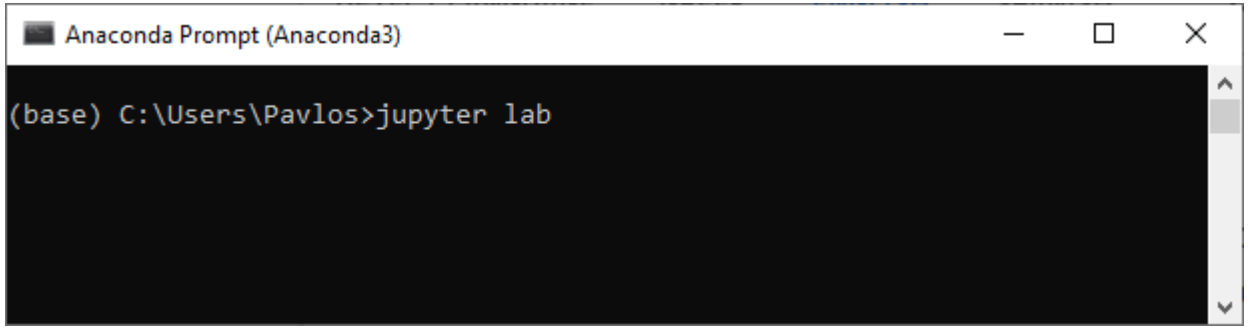


Launch Jupyter Lab from the command line



- Launch Anaconda Prompt in Windows → (or terminal in Linux / macOS)

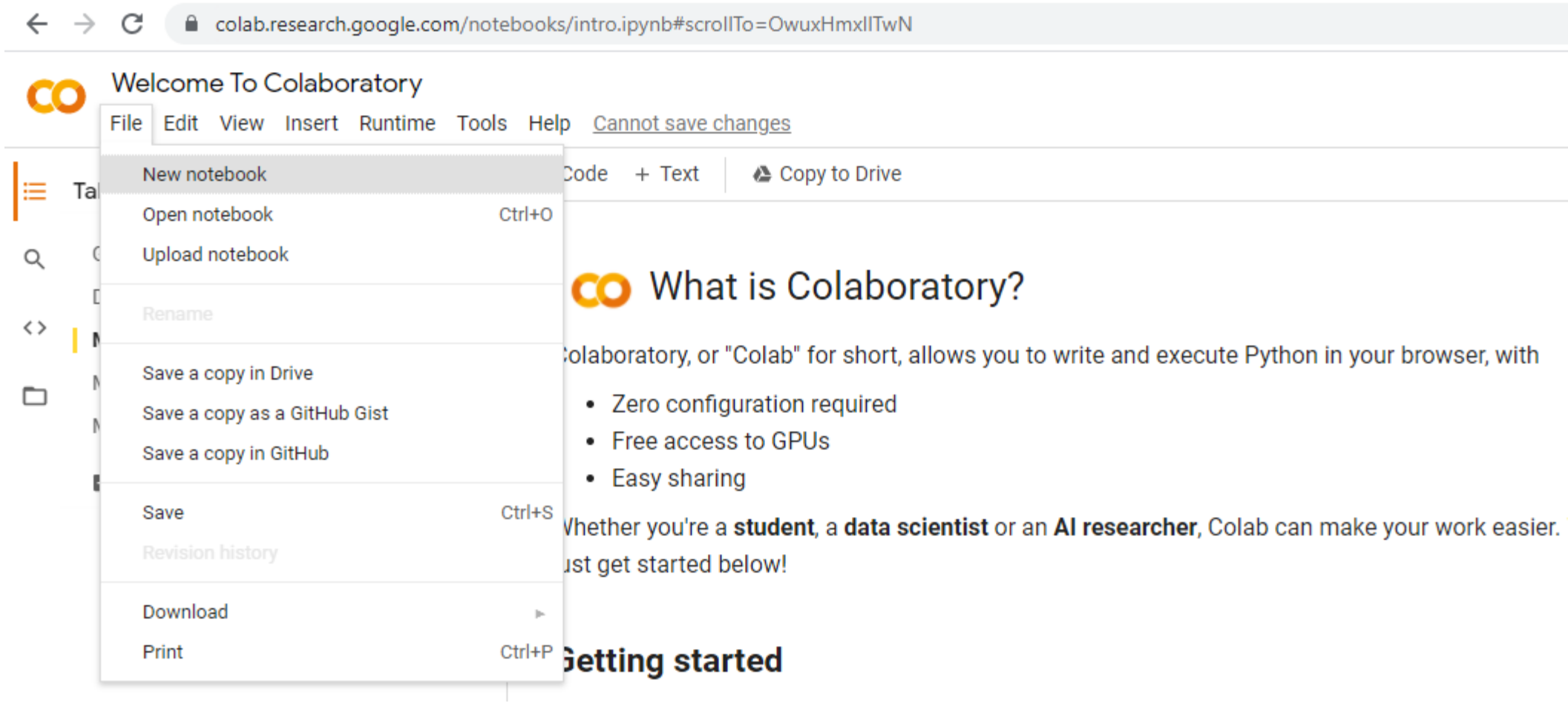
- Type `jupyter lab` and press Enter
  - Jupyter Lab should start up just like it did when you launched it from Anaconda Navigator by via a new browser window



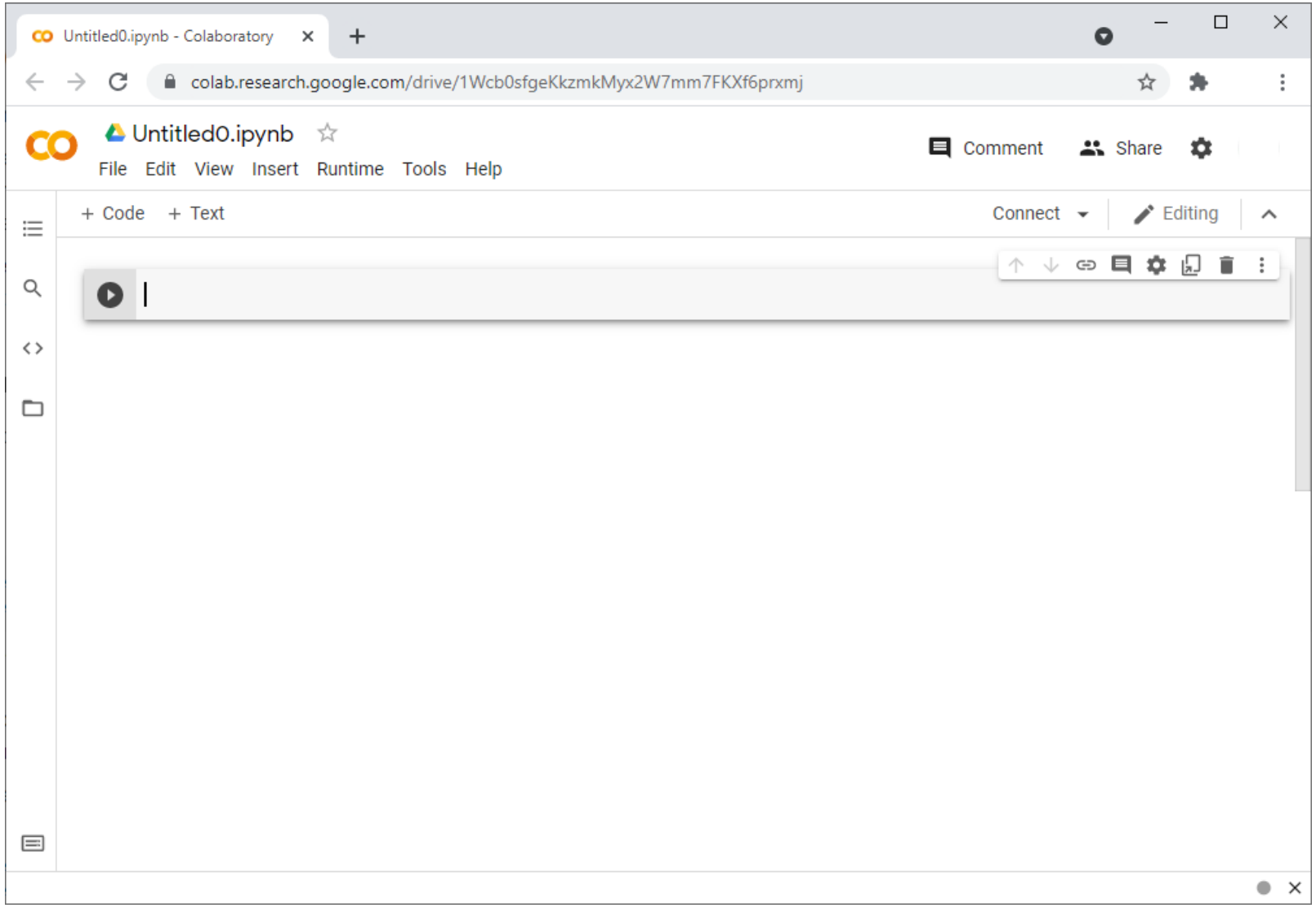
Google Colab or Colaboratory

- Freemium product by Google research, based on Jupyter
- Accessed via browser like Jupyter but connection to Internet needed
- No need to install anything on your machine (e.g. Anaconda, JupyterLab, etc.) – runs on Google servers
- Free computation power (GPU and TPU)
- Almost all important libraries are pre-installed
- Colab's notebook files are stored in your google drive, so they can be accessed from anywhere
- Allows for sharing notebooks with other people without even downloading them
- Start using Google colab: https://colab.research.google.com/
- Disadvantages: Limited resources (RAM, CPU) unless you use the paid version, dependency on internet connection.

Google Colab Notebook



Google Colab Notebook



A Code Sample

```
x = 34 - 23                    # A comment.
y = "Hello"                    # Another one.
z = 3.45
if z == 3.45 or y == "Hello":      # colon needed
    x = x + 1                  # similar to x += 1.
    y = y + " World"           # String concatenation.
print(x)                       # 12
print(y)                       # Hello World
x = y
print(x)                       # Hello World
```

Enough to Understand the Code

- Assignment uses `=` and comparison uses `==`
- For numbers, arithmetic operators `+ - * / %` are as expected
  - Special use of `+` for string concatenation: `"Hello" + "World"`
  - Special use of `%` for string formatting (see next slides)
- Logical operators are words (`and`, `or`, `not`) **not** symbols
- The basic printing command is `print()`
- The first assignment to a variable creates it e.g.: `x = 8`
  - Variable types don't need to be declared
  - Python figures out the variable types on its own
  - Multiple assignment is also available e.g.: `x, y = 2, 3`

Arithmetic Operators

Operator	Name	Examples
+	Addition	`3 + 5 returns 8` `"a" + "b" returns "ab"`
-	Subtraction	`50 - 24 returns 26`
*	Multiplication	`2 * 3 returns 6` `"la" * 3 returns "lalala"`
**	Exponentiation	`3 ** 4 returns 81 (i.e. 3 * 3 * 3 * 3)`
/	Division	`4 / 3 returns 1.3333333333333333`
//	Floor Division	`4 // 3 returns 1` `5.4 // 2.1 returns 2.0 (5.4/2.1 → 2.5714285714285716)`
%	Modulus	`8 % 3 returns 2` `-25.5 % 2.25 returns 1.5`

Multiple ways of printing

```
a = 10
b = 20
c = a + b

# Space is automatically printed in the position of each comma
print("sum of", a , "and" , b , "is" , c) # sum of 10 and 20 is 30

# Create a long string and print it. All non string variables must be
converted into string with str(). Spaces must be defined explicitly.
print("sum of " + str(a) + " and " + str(b) + " is " + str(c))

# if you want to print in tuple way (C-like way)
print("sum of %d and %d is %d" %(a,b,c))

# New style string formatting
print("sum of {0} and {1} is {2}".format(a,b,c))
```

Numeric Datatypes

- Integer numbers (int)
  - `z1 = 23`
  - `z2 = 5 // 2  # Answer is 2, integer division.`
  - `z3 = int(6.7)  # Converts 6.7 to integer. Answer is 6.`
  - Booleans (bool) are a subtype of integers
- Floating (float) point numbers (implemented using double in C)
  - `x1 = 3.456`
  - `x2 = 5 / 2  # Answer is 2.5`
  - `x3 = float(4)  # Answer is 4.0`
- Complex numbers
- Additional numeric types: fractions, decimal
- See more here

Newlines and Whitespaces

- Use a newline to end a line of code.
  - Use \ when must go to next line prematurely.
- Whitespace is meaningful in Python: especially indentation
- No braces { } to mark blocks of code in Python…
  Use consistent indentation – whitespace(s) or tab(s) – instead.
  - The first line with more indentation starts a nested block
  - The first line with less indentation is outside of the block
  - Indentation levels must be equal within the same block but not necessarily the same with other blocks

```
if x%2 == 0:
    print("even")
    print("number")
else:
 print("odd")
```

- Often a colon appears at the start of a new block.
  - e.g. in the beginning of `if`, `else`, `for`, `while`, as well as function and class definitions

Comments

- Single line comments: Start comments with # – the rest of line is ignored by the python interpreter

```
# this is a single-line comment
```

- Multiple line comments: Start/end comments with """

```
"""
this is
a multi-line
comment
"""
```

Accessing Non-Existent Names

- If you try to access a variable name before it's been properly created (by placing it on the left side of an assignment), you'll get an error.

```
>>> print(y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
>>> y = 3
>>> print(y)
3
```

Naming Rules

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.

```
bob Bob _bob _2_bob_ bob_2 BoB
```

- There are some reserved words:

```
and, assert, break, class, continue, def, del,
elif, else, except, exec, finally, for, from,
global, if, import, in, is, lambda, not, or,
pass, print, raise, return, try, while
```

Some Python datatypes (objects)

- Some **immutable** objects
  - **int**
  - **float**
  - decimal
  - complex
  - bool

  Numeric datatypes
  - **string**
  - **tuple**
  - bytes
  - **range**

  Sequences
  - frozenset

  Set type
- Some **mutable** objects
  - **list**
  - bytearray

  Sequences
  - set

  Set type
  - **dict**

  Mapping
  - user-defined classes (unless specifically made immutable)

- ❖ When we change these data, this is done in place.
- ❖ They are not copied into a new memory address each time.

values cannot be changed in the same place in memory

Immutable Sequences I

- **Strings**
  - Defined using double quotes `""` or single quotes `''`

    ```
    >>> st = "abc"
    >>> st = 'abc' (Same thing.)
    ```

  - Can occur within the string.

    ```
    >>> st = "matt's"
    ```

  - Use triple double-quotes for multi-line strings or strings than contain both ‘ and “ inside of them:

    ```
    >>> st = """This is a multi-line
    string that uses triple quotes."""
    >>> st = """a'b"c"""
    ```

Immutable Sequences II

- **Tuples**
  - A simple **immutable** ordered sequence of items of mixed types
  - Defined using parentheses (and commas) or using `tuple()`.

```
>>> t = tuple()          # create empty tuple
>>> tu = (23, 'abc', 4.56, (2,3), 'def')   # another tuple
>>> tu[2] = 3.14
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

  - You can't change a tuple.
  - You can make a fresh tuple and assign its reference to a previously used name.

```
>>> tu = (23, 'abc', 3.14, (2,3), 'def')
```

Immutable Sequences III : data access

- We can access individual members of a **tuple** or **string** using square bracket “array” notation.
- Positive index: count from the left, starting with 0.
- Negative index: count from right, starting with –1.

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1] # Second item in the tuple.
'abc'
>>> tu[-3]
4.56
>>> st = "Hello World"
>>> st[1] # Second character in string.
'e'
```

Mutable Sequences I

- **Lists**
  - **Mutable** ordered sequence of items of mixed types
  - Defined using square brackets (and commas) or using `list()`.

```
>>> li = ["abc", 34, 4.34, 23]
```

  - We can access individual members of a list using square bracket “array” notation as in tuples and strings.

```
>>> li[1] # Second item in the list.
34
```

  - We can change lists in place.
    - Name li still points to the same memory reference when we’re done.
    - The mutability of lists means that they aren’t as fast as tuples.

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

Tuples vs. Lists

- Lists slower but more powerful than tuples.
  - Lists can be modified, and they have lots of handy operations we can perform on them.
  - Tuples are immutable and have fewer features.
- To convert between tuples and lists use the list() and tuple() functions:

```
li = list(tu)
tu = tuple(li)
```

Slicing in Sequences: Return Copy of a Subset 1

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- Return a **copy** of the container with a subset of the original members. Start copying at the first index, and stop copying before the second index.

```
>>> tu[1:4]
('abc', 4.56, (2,3))
```

- You can also use negative indices when slicing.

```
>>> tu[1:-1]
('abc', 4.56, (2,3))
```

Slicing in Sequences: Return Copy of a Subset 2

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- Omit the first index to make a copy starting from the beginning of the container.

```
>>> tu[:2]
(23, 'abc')
```

- Omit the second index to make a copy starting at the first index and going to the end of the container.

```
>>> tu[2:]
(4.56, (2,3), 'def')
```

Copying the Whole Sequence

- To make a copy of an entire sequence, you can use [:].

```
>>> tu[:]
(23, 'abc', 4.56, (2,3), 'def')
```

- Note the difference between these two lines for mutable sequences:

```
>>> list2 = list1        # 2 names refer to 1 reference
                         # Changing one affects both
>>> list2 = list1[:]     # Two independent copies, two refs
```

The ‘in’ Operator

- Boolean test whether a value is inside a container:

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

- For strings, tests for substrings

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

- Be careful: the `in` keyword is also used in the syntax of for loops and list comprehensions.

Operations on Lists Only 1

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

Operations on Lists Only 2

- extend operates on list li in place (adds the elements of a list to another list)

```
>>> li.extend([9, 8, 7])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

- Confusing:
  - Extend takes a list as an argument.
  - Append takes a singleton as an argument.

```
>>> li.append([10, 11, 12])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [10, 11, 12]]
```

Operations on Lists Only 3

```
>>> li = ['a', 'b', 'c', 'b']
>>> li.index('b')       # index of first occurrence
1
>>> li.count('b')       # number of occurrences
2
>>> li.remove('b')      # remove first occurrence
>>> li
['a', 'c', 'b']
```

Operations on Lists Only 4

```
>>> li = [5, 2, 6, 8]
>>> li.reverse()  # reverse the list *in place*
>>> li
[8, 6, 2, 5]
>>> li.sort()     # sort the list *in place*
>>> li
[2, 5, 6, 8]
```

Dictionaries: A Mapping type

- Dictionaries store a mapping between a set of keys and a set of values.
  - Dictionaries are **mutable**
  - Keys can be any immutable type.
  - Values can be any type
  - A single dictionary can store values of different types
  - Defined using { } : (and commas).
- You can define, modify, view, lookup, and delete the *key-value* pairs in the dictionary.

Using dictionaries

```
d = {'user': 'john', 'pswd': 1234}
print(d['user'])      # john
print(d['pswd'])      # 1234
print(d['john'])      # KeyError: 'john'
d['user']='bill'      # modify user value
print(d)              # {'user': 'bill', 'pswd': 1234}
d['id']=45            # add another key
print(d)              # {'user': 'bill', 'pswd': 1234, 'id': 45}
del d['user']         # remove one key/value pair
print(d)              # {'pswd': 1234, 'id': 45}
                      # d.clear() to remove all key/value pairs
print(d.keys())       # dict_keys(['pswd', 'id'])
print(d.values())     # dict_values([1234, 45])
```

Control of flow 1

- The `if`/`elif`/`else` statement

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

Control of flow 2

- The `while` statement

```
x = 0
while x < 5:
    print(x)
    x = x + 1
print("Outside of the loop.")
```

Output:

0
1
2
3
4
Outside of the loop.

Control of flow 3

- The `for` statement

```
                          # the same as
for i in range(5):        # for i in [0,1,2,3,4]:
    print(i)
print("Outside of the loop.")
```

Output:

```
0
1
2
3
4
Outside of the loop.
```

range()

- The range() function has two sets of parameters, as follows:
  - range(stop)
    - stop: Number of integers (whole numbers) to generate, starting from zero.
      E.g. `range(3)` → `[0, 1, 2]`
  - range([start], stop[, step])
    - start: Starting number of the sequence.
    - stop: Generate numbers up to, but not including this number.
    - step: Difference between each number in the sequence.

    E.g. `range(10,2,-2)` → `[10, 8, 6, 4]`
- Note that:
  - All parameters must be integers.
  - All parameters can be positive or negative.

Control of flow 4

- The `for` statement

```
for i in [3, 6, 9]:
    print(i)
```

Output:

3
6
9

```
for c in "Hello":
    print(c)
```

Output:

H
e
l
l
o

User-defined functions

- `def` creates a function and assigns it a name
- `return` sends a result back to the caller

```
def <name>(arg1, arg2, ..., argN):
    <statements>
    return <values>
```

```
def times(x,y):
    return x*y
```

Function call:

```
x = times(4,5) # returns 20
```

Optional Arguments

- Can define defaults for arguments that need not be passed

```
def func(a, b, c=10, d=100):
    print(a,b,c,d)

>>> func(1,2)
1 2 10 100
>>> func(1,2,3,4)
1 2 3 4
```

Built-in functions

- https://docs.python.org/3/library/functions.html

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

`len()` :

- Return the length (the number of items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

`min()` / `max()` :

- Return the smallest / largest item in an iterable or the smallest of two or more arguments.

Built-in functions: len(), max(), min()

```
>>> my_list = ['one', 'two', 3]
>>> my_list_len = len(my_list)
>>> for i in range(0, my_list_len):
...     print(my_list[i])
...
one
two
3
>>> max("hello","world")
'world'
>>> max(3,13)
13
>>> min([11,5,19,66])
5
```

Modules

- Modules are functions and variables defined in separate files
- Items are imported using from or import

```
from module import function
function()
```

A' Way

```
import module
module.function()
```

B' Way

Mathematical functions

- https://docs.python.org/3.9/library/math.html

```
>>> import math
>>> print(math.sqrt(3))
1.7320508075688772

>>> from math import sqrt
>>> print(sqrt(3))
1.7320508075688772
```

Lambda function - a quicker way of writing functions on the fly

- Shorthand version of def statement; Useful for “inlining” functions
- A lambda function can take any number of arguments, but can only have one expression (e.g., no if statements, etc)
- A `lambda` returns a function; the programmer can decide whether or not to assign this function to a name
- Simple example:

```
>>> def sum(x,y): return x+y
>>> sum(1,2)
3
>>> sum2 = lambda x, y: x+y
>>> sum2(1,2)
3
```