

# Lecture 1: Python Introduction

## Goals for Today:

### Jupyter Notebooks:

- Installation options
- Basic functionality

### Python:

- Basic syntax
- Data types
- Operators
- Data Structures



# Jupyter Notebooks

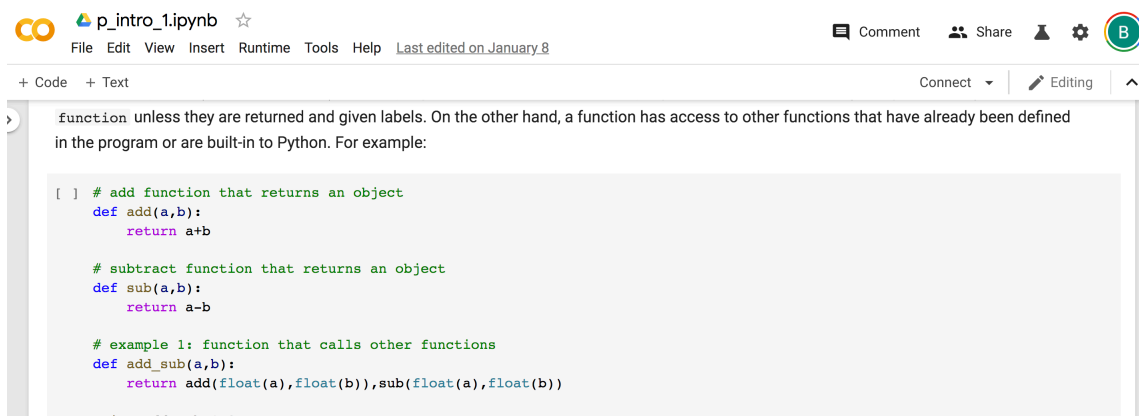
All programming for the course will be done using **Jupyter Notebooks**.

**Notebooks** are a convenient way of interspersing executable code, results, and text in a reproducible format. If you've ever used Maple or Mathematica, that's the idea.

**Jupyter** is a popular open-source framework for hosting python-based notebooks (and other languages) in web browsers. Jupyter will allow us to write Python code and distribute the results in these self-contained notebooks.



## Example from first week's notes:



```
[ ] # add function that returns an object
def add(a,b):
    return a+b

# subtract function that returns an object
def sub(a,b):
    return a-b

# example 1: function that calls other functions
def add_sub(a,b):
    return add(float(a),float(b)),sub(float(a),float(b))

print(add_sub(1,2))
```

The Python language is independent of Jupyter. I don't want you to think that you always need a notebook for writing and executing code. But for this course, we will always do our work in the context of Jupyter.

# Jupyter Notebooks: Option 1

For this course there are three ways for you to open and write Jupyter notebooks:

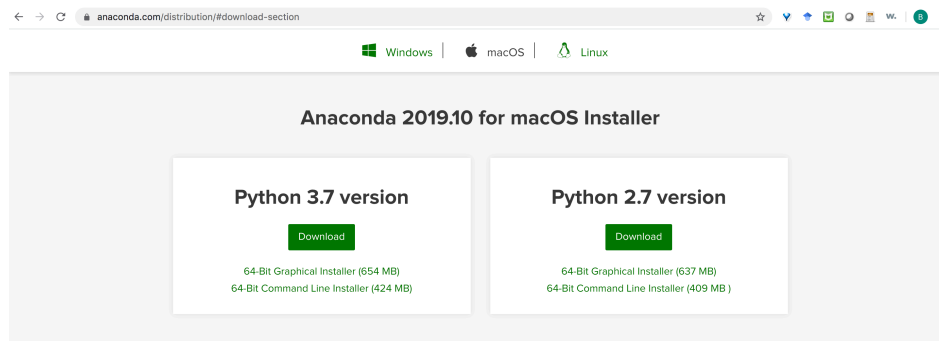
## Option 1: Anaconda

Continuum analytics distributes a popular Python 3.x distribution for scientific computing called “**Anaconda**”. This option will allow you to easily install common modules and run Jupyter without an internet connection on your home computer.



### Steps:

- 1) Download the python 3.7 version of anaconda for your operating system at <https://www.anaconda.com/distribution/>



- 2) You can launch jupyter using anaconda’s graphical interface or by running “jupyter” in your system’s command-line. See here for additional os-specific details: <https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/execute.html>

# Jupyter Notebooks: Option 2

For this course there are three ways for you to open and write Jupyter notebooks:

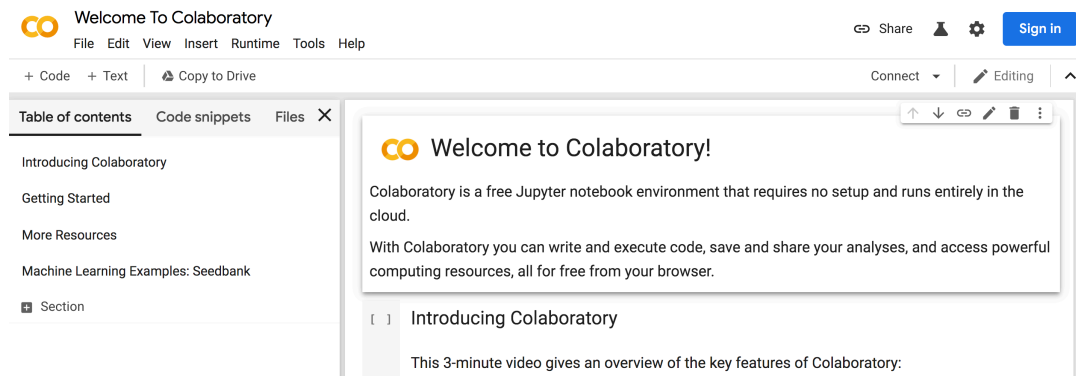
## Option 2: Google's Colaboratory

Google currently hosts a free jupyter notebook environment on the cloud call colaboratory. Using colab, you don't need to install anything, just create a google account. For now, google provides free cpu and gpu resources to the users of colab.



### Steps:

- 1) Navigate your favorite browser to <https://colab.research.google.com/>



- 2) You'll need to make an account if you don't already have one so that you can save your notebooks on your google drive. Explore the details of loading and saving notebooks using google's tutorials.

# Jupyter Notebooks: Option 3

For this course there are three ways for you to open and write Jupyter notebooks:

## Option 3: Purdue Scholar

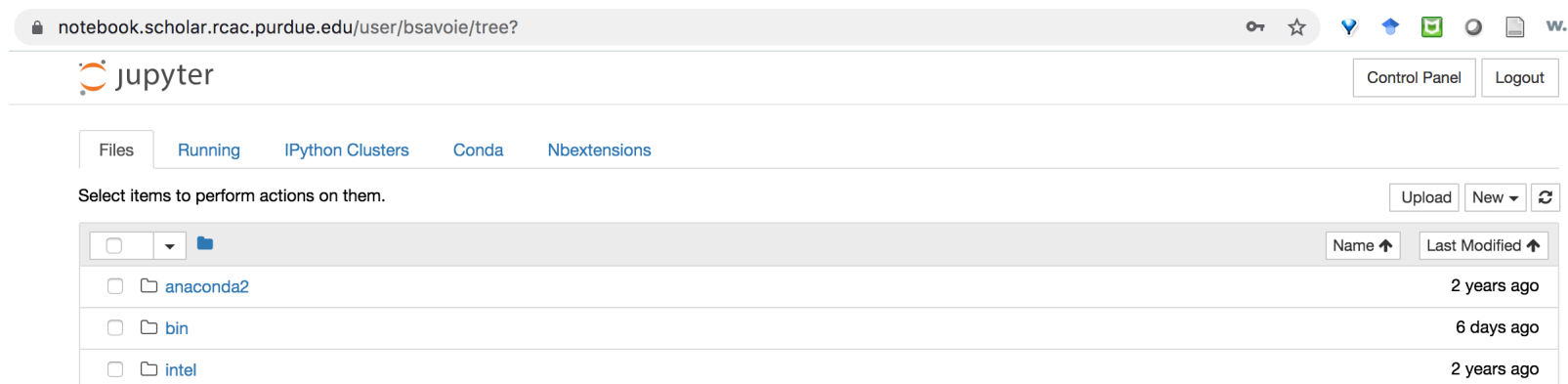
ITAP hosts Jupyter notebooks on the academic cluster Scholar. You can do many of the class assignments on scholar, but the environment doesn't come equipped with all of the packages we will use, so will need to load modules to get access to some libraries. See here for more details:

<https://www.rcac.purdue.edu/knowledge/scholar/jupyter>



### Steps:

1) Navigate your favorite browser to <https://notebook.scholar.rcac.purdue.edu/>



2) You'll be asked to login, then you'll land on a page similar to the above from which you can create folders, upload, and run notebooks.

# Your First Python Program

```
a = 1  
b = 2  
print(a+b)
```

```
3
```

Looking at the output, nearly everyone can piece together what has happened, but a lot of complexity goes into making things seem this simple:

*For instance, we created two integers in the system's memory (1 and 2). We didn't tell the computer where to put those, we left it up to Python's **interpreter** (a behind the scenes compiler) and the system's **operating system** (which actually executed the compiled program) to just figure out whether to put those in **RAM** or on the **disk drive**, and to do it in a manner that didn't conflict with other things that might be saved in those places.*

# Your First Python Program

```
a = 1  
b = 2  
print(a+b)
```

```
3
```

Looking at the output, nearly everyone can piece together what has happened, but a lot of complexity goes into making things seem this simple:

*We also attached labels to those integers ( $a$  and  $b$ ), so that we could use them later. Creating **pointers**, or giving names to objects in a program, is something that comes naturally to us, but behind the scenes the **compiler** needs to do more hard work to provide this abstraction.*

# Your First Python Program

```
a = 1  
b = 2  
print(a+b)
```

```
3
```

Looking at the output, nearly everyone can piece together what has happened, but a lot of complexity goes into making things seem this simple:

*How did the computer know that 1 and 2 were integers? How did it know that we didn't mean 1.0 and 2.0 (**floats**), or "1" and "2" (**strings**)? That's a convention of the language and you have to play by its rules. In Python, a number without a decimal is interpreted as an integer and anything inside of single ( ' ') or double ( " " ) quotes is a string. In other languages you need to be more explicit.*



# Your First Python Program

```
a = 1  
b = 2  
print(a+b)
```

```
3
```

Looking at the output, nearly everyone can piece together what has happened, but a lot of complexity goes into making things seem this simple:

*How did the computer know what  $a+b$  meant? There are a lot of assumptions we made there without thinking about it. We assumed that the Python knows how to interpret  $+$  (an **operator**). Thankfully it does. We also assumed that it would know that  $a$  and  $b$  pointed to things that could be added. What if  $a$  and  $b$  had been strings?*

# Your Second Python Program

```
a = "one"  
b = "two"  
print(a+b)
```

```
onetwo
```

*In this case the program interpreted + in different way, to mean "add these strings together". This is again something built into the language that we can take for granted in Python but will be different in other languages.*

# Your Second Python Program

```
a = "one"  
b = "two"  
print(a+b)
```

```
onetwo
```

*What is `print()`? I think everyone can guess that `print()` "prints" whatever is inside its parenthesis, but how did the program know it could do this? There are a lot of capabilities that are built into every modern programming language. For instance, all modern programming languages assume every user will potentially need to do things like print out results (`print()`) and add (+). Some capabilities aren't available by default and we'll need to alert the program that we want to use them (**importing**).*

# Your Second Python Program

```
a = "one"  
b = "two"  
print(a+b)
```

```
onetwo
```

At what point did the human-readable commands above (**source-code**) get turned into machine code? Part of the reason that Python is so useful is because this intermediate step is handled behind the scenes by an **interpreter** and a specialized Python **compiler**. In other languages, compiling source-code is a separate step and you need to be careful how you do it. In Python, this is handled behind the scenes for you. This is why Python is sometimes called an **interpreted** language versus a **compiled** language. But these distinctions and how they apply to Python are way beyond what you need to know right now. This is the last you will hear of **compilers** and **interpreters** in this course because we need to move on to doing things.

# Your Second Python Program

```
a = "one"  
b = "two"  
print(a+b)
```

one two

We've covered a lot of concepts already. In the rest of this lecture we'll put each topic on a firm footing and incrementally build up our knowledge of the basic building blocks of Python code.

At w  
code  
Pyth  
behin  
com  
step  
hand  
some

source-  
that  
ndled  
ython  
arate  
his is  
on is  
versus

a **compiled** language. But these distinctions and how they apply to Python are way beyond what you need to know right now. This is the last you will hear of **compilers** and **interpreters** in this course because we need to move on to doing things.

# Pointers vs. Objects in Memory

```
a=1  
a='one'  
print(a)
```

```
one
```

In a dynamically typed language, **pointers** are labels for objects in memory (like 1 above). They can be plucked off one object and placed on another.

the program first creates an integer in memory corresponding to 1, with the pointer `a` pointing to it. After the second line, a string `'one'` is created in memory and the pointer `a` now points to it.

If the 1 initialized in memory isn't used for a while, a **garbage collector** comes around and automatically deletes it from memory. In languages like C it is up to the user to allocate and release memory, but in Python the **garbage collector** (this is actually what it is called) periodically deletes things once the program you are executing no longer references them.

# Strong Dynamic Typing

```
a=1  
b=a+1  
print(b)
```

2

The **type** of 1 was determined to be an integer on the fly. We call this aspect of a language "**dynamic typing**" and it means that the user doesn't need to explicitly specify the **type** of an object. This can be a surprise for someone coming from the C-family or Fortran programming languages.

```
a=1  
b=a+"one"
```

```
TypeError    Traceback (most recent call last)  
<ipython-input-22-265f3fb8a2e5> in <module>  
      1 a=1  
----> 2 b=a+"one"
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Even though Python determines object **types** dynamically when they are created, the object **type** is fixed. We call this aspect of a programming language **strong typing**. In practical terms, all that this means is that Python will throw an error if you try and do something with an object that it is incapable of.

# Data Types

We'll refer to **objects** as anything that is stored in memory and **data** objects as special types of objects that are single values. Python handles several data types natively:

- int**: integers are whole numbers (no decimal places). Python automatically interprets any number specified without a decimal place as an integer (e.g., `1`, `3`, `100001`, and `0` are all integers).

- float**: floats are decimal numbers. Python automatically interprets any number specified with a decimal as a float (e.g., `1.`, `1.0`, and `1.00000` all refer to the same float). Behind the scenes floats are stored as the closest binary fraction of a fixed length.

- string**: strings are words, sentences, or anything that you want the computer to interpret "literally". Python automatically interprets any characters specified inside single (`' '`) or double (`" "`) quotes as a string (e.g., `"blue"`, `"words are great"`, and `"1.0"` are all strings).

- boolean**: boolean types are things that have only two values, `True` and `False`. In a lot of applications we need booleans, or at least use them behind the scenes to evaluate whether or not something is true. For example, `1<2` returns the boolean `True`, and `1>2` returns the boolean `False` in Python. `True` and `False` are special values in Python reserved for booleans.

- complex**: complex types are imaginary numbers. Python automatically interprets any number written as `a + bJ` as a complex type, where `a` and `b` can be integers or floats (e.g., `1.0+1.0J`, `1+1J`, and `1 + 1.0J` all specify complex numbers with a real and imaginary component of 1).



# Data Types: Examples

```
# Example 1: integer math
a = 1000
b = 2000
print(a+b)

# Example 2: float math
a = 1000.0
b = 2000.0
print(a+b)
print(type(a+b))
```

```
3000
3000.0
<class 'float'>
```

In the first example, adding two `int` objects returns an `int` result

In the second example, adding two `float` objects returns a `float` result.

We can also confirm the data types by printing out the `type()` explicitly.

# Data Types: Upcasting

```
# Example 1: upcasting, integer math
a = 1
b = 3
print (a/b)

# Example 2: upcasting, mixed integer/float math
a = 1000
b = 2000.0
print (a+b)
print (b/a)
```

```
0.333333333333
3000.0
2.0
```

In all three cases we obtained float results. This is expected behavior known as **upcasting**.

When you ask Python to do something with an object, it will *sometimes* change the object's type (if it can) in order to do what you asked.

# Data Types: Explicit Casting

```
# Example 1: Explicit casting, integer math
a = 2
b = 3
print(int(a/b))

# Example 2: Explicit casting, float math
a = float(2)
b = float(3)
print(a/b)

# Example 3: complex math
a = 1.0 + 1.0J
b = 1.0 + 5.0J
print(a+b)

# Example 4: string concatenation
a = "1.0"
b = '3.0'
print(a+b)
```

```
0
0.666666666666
(2+6j)
1.03.01
```

We can use the built-in functions `int()` and `float()` to explicitly cast (or force) the data type of each object and result.

# Operators

**Operators** are specialized commands that “operate” on objects to their left and right and return specialized values.

**Operators** can be defined as single characters (e.g., +, -, /, \*, %, >, <) or multiple characters (e.g., ==, is, >=, \*\*)

**Logical** operators (e.g., is, >, <, ==, etc) evaluate to True or False.

```
a=1
b=1
c=True

# Example 1: logical "==" operator
print(a == b)

# Example 2: logical "is" operator
print(c is True)

# Example 3: logical ">" operator
print(100>1)

# Example 4: logical "<" operator
print("abce"<="abcd")
```

```
True
True
True
False
```

# Operators

**Operators** are specialized commands that “operate” on objects to their left and right and return specialized values.

**Operators** can be defined as single characters (e.g., +, -, /, \*, %, >, <) or multiple characters (e.g., ==, is, >=, \*\*)

The operators we’ve just used are all built-in to Python (i.e., they are defined by default). You can “override” or redefine these operators so that they behave differently, but this is an advanced topic that isn’t necessary for the course.

```
True
True
True
False
```

# Functions: built-in

A **function** is a special type of object that takes other objects as input and does something or optionally **returns** other object(s) as output(s).

**functions** allow you to reuse chunks of code for operation on new inputs or with tweaks to the parameters associated with the reused code.

A **function** is called in Python by using its name followed by parentheses (e.g., `print(1)` calls the `print()` function with the integer 1). Most, though not all, functions will accept some kind of object as an input. The inputted objects to a function are called its **arguments**, and these go inside the parentheses.

```
# example: calling the print() function on an integer
a=print(1)

# print() does not return an object
print(a)
```

```
1
None
```

Some functions **return** new objects, some do not. `print()` does not.

# Functions: custom

**return:** When we say that a function **returns** something, we mean that the function creates an object that we intend to keep for use in the rest of the program. Some functions, like `print()`, do not **return** objects. Some functions **return** several objects.

**arguments:** the objects that we pass to a function are its **arguments**. These can be optional or required depending on the function. When multiple arguments are supplied they are separated by a comma (,).

We can define our own functions using the keyword `def`

```
# example: function that returns an object
def add(a,b):
    return a+b
print(add(1,3))
```

4

We have defined a simple `add` function that has `a` and `b` as inputs and **returns** `a+b` as an output. `def name(input1, input2, ...):` is the syntax in Python for defining a function.

# Functions: custom

let's reformulate the example so that the second argument `b` is **optional**. In this case, it acts the same as the original function when two arguments are supplied, but defaults to adding 0 when only one argument is supplied.

```
# example: add function with an optional second argument (a is
positional, b is optional)
def add(a,b=0):
    return a+b

# print results
print(add(1,3))
print(add(1,b=3))
print(add(1))
```

```
4
4
1
```

Required arguments are positional, meaning that they have to be supplied in order (e.g., `a` above), optional arguments can be supplied using `name=object` syntax or positionally.



# Functions: using returned objects

Functions can return multiple objects that can be assigned to pointers when called.

```
# example: function that returns multiple objects
def add_sub_mult_div(a,b):
    return a+b,a-b,a*b,a/b

# label results for reuse
a,b,c,d = add_sub_mult_div(1.,3.)
print("a: "+str(a))
print("b: "+str(b))
print("c: "+str(c))
print("d: "+str(d))
```

```
a: 4.0
b: -2.0
c: 3.0
d: 0.333333333333
```

We can assign a pointer to returned objects by calling the function with an equality.

Multiple pointers are assigned by separating the pointers by commas.

# Indenting

In this example, how did the python interpreter know where the function ended?

```
# example 2: function that returns multiple objects
def add_sub_mult_div(a,b):
    return a+b, a-b, a*b, a/b
```

In most programming languages you would use a “start” keyword, like `def`, and a separate “end” keyword to let the compiler know where the function ends.

In Python, instead of using “end” keywords for the various situations that require them, you instead use indenting to specify when code is part of the function and when it is part of the rest of the program.

Indenting is used in this way anytime you are doing something that seems to logically require an “end” statement (e.g., this will figure prominently in the control statements section).

# Namespace

```
# example: function that returns multiple objects
def add_sub_mult_div(a,b):
    return a+b,a-b,a*b,a/b

# label results for reuse
a,b,c,d = add_sub_mult_div(1.,3.)
print("a: "+str(a))
print("b: "+str(b))
print("c: "+str(c))
print("d: "+str(d))
```

In the examples above, inside the functions the labels `a` and `b` have one meaning, but outside the functions, we use `a` and `b` to label outputs. How come these don't conflict in some way?

**Namespace** is the term used for how programming languages organize pointers. The short answer is that the names for things inside of functions are actually renamed internally by Python without you knowing it in order to make sure that they don't conflict with labels in the rest of the code.

You can make an object "**global**" so that it is accessible everywhere in your code, but this shouldn't be done naively and is beyond the scope of the course.

# Scope

```
# add function that returns an object
def add(a,b):
    return a+b

# subtract function that returns an object
def sub(a,b):
    return a-b

# example 1: function that calls other functions
def add_sub(a,b):
    return add(float(a), float(b)), sub(float(a), float(b))

print(add_sub(1,3))
```

```
(4.0, -2.0)
```

**scope** refers to which objects can be accessed by different parts of your program.

For example, a **function** only has access to data objects that are passed as inputs, data objects that are defined in the function itself, and other functions that have been defined in the program (e.g., above).

Likewise, the rest of the program cannot access the objects generated by a function unless they are returned and given pointers.

# Data Structures

In any practical application it will quickly become inconvenient to work with objects on an individual basis and we will want to collect objects together--that's what **data structures** are for:

- list**: lists are the structures that store data in **order**. Lists have built-in methods for adding more data, removing data, modifying data, and accessing data in the list. You can modify objects in the list without creating a whole new list, for this reason we say that lists are **mutable**. Defined using `[]` notation.

- tuple**: tuples are structures that store data in **order** but are **immutable**. That is tuples are like lists, except that you can't add or remove from them without creating a whole new object in memory. Sometimes you need things that are immutable (e.g., the next two structures require immutable objects!). Defined using `()`.

- set**: sets are like lists but they are **unordered** and no duplicates are allowed. Unordered? Yes, sets collect objects but don't keep track of where they were put. On a technical side, it is actually more accurate to say that sets are hyperspecific about where they've put things (internally) and so you as the user cannot modify that order. This makes sets very efficient at calculating membership (e.g., is "1" in our set?) and intersections (common objects in multiple sets). Defined using `{}` or `set(list)` notation.

- dictionary**: dictionaries are structures that store objects in pairs. Each pair consists of a **key** and a **value**. In analogy, a phone book is a dictionary, where the key is a person's name, and the value their phone number. You could accomplish the same thing using two sets of ordered lists, but dictionaries have the advantage that their keys are stored...as...a...**set**! Defined with `{}` notation with key:value pairs. (check examples below)