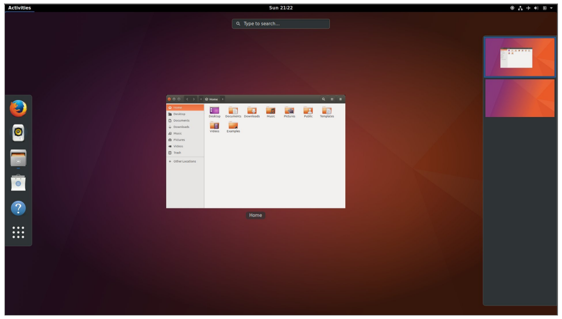
**Lecture 1**: Python Variables, Expressions, Data Types, Lists

# Our Conscious Experience of a Computer



- When we work on a computer, the experience is not veridical or a "true" representation of reality.
- No matter what type of computer (Windows, Mac, Linux) we are always interacting with an interface which contains similar elements.
- The Desktop is a graphical representation of your computer. It varies a bit between different types of computers but contains familiar elements - files, folders, applications.
- None of these elements are "real". For example, a file is not represented in the computer in the manner of the icon on the Desktop.

## Programs are Interfaces to the Computer - **You** give them meaning.

- The Desktop is an interface to allow you to manipulate the computer to achieve your objectives, without knowing too much about the inner workings of the computer.
- **Programming is an interface to allow you to instruct the computer to manipulate information stored in the memory of the computer.**
- Programming is a way to ask the computer to store values (**variable**) and do things with them (**operations**)
- The elements and rules of programming (**syntax**) are just like "rules" of working with the desktop of the computer
- They are a representation in a human readable language of an operation in the computer language - a good program tells a story.

# Variables - Definitions

1. A variable is a symbolic representation of the **location** of information in the memory of a computer.
2. A variable is a way to address and manipulate the memory of the computer using a label that ascribes meaning to the information stored there.
3. Naming variables in meaningful ways is perhaps the most important thing to learn to write **readable** programs.

In [48]:

```
#Remember, comment lines start with a #
#Comments are useful to track what you are doing.
#But, too many comments means you didnt write a self-explanatory program.

my_variable = 1 # my_variable is a variable I created with a statement
my_computed_variable = 4/5 #This is a variable I computed with an operation
my_text_label = 'Example' #This is a bit of text
```

- There are three types of variables created above.

1. numeric integer **my_variable** which is of type **int**
2. numeric floating point **my_other_variable** which is of type **float**
3. text string **my_label** which is of type **str**

- Lets Examine the Variables pane to confirm my definitions.
- These are 3 of the 4 basic types of variables. The one missing here is a *logical* variable, which will be discussed later.

## Expressions - Definitions

1. An expression is an operation on variables. It may be used to define a new variable.
2. In scientiifc applications, an operation is often a mathematical statement.

In [49]:

```
x = 2 #This is a statement that declares a variable with a particular value.
y = x**2 #This is a mathematical statement written as code.
print(x)
print(y)
```

```
2
4
```

- In the example above a mathematical expression, $y = x^2$ is written as code.

- Consider the following 2 lines of code. Here code is not strictly math.

In [50]:

```
z = 3 # a variable you created
z = z**2 #This is code, not math.
```

- we are again computing the square of a number, but we are telling python to **replace** the value of $z$ with $z^2$ in the **location of memory** we address with $z$.

## Key Concepts

- In programming `=` means *assignment*, not equality
- Anything to the right of the equality is evaluated before assignment
- There can be more than one variable assigned in a single line.

## Mathematical Operators

1. + , addition
2. - , subtraction
3. * , multiplication
4. / , division
5. ** , exponentiation
6. / / , floor division or integer division
7. %, remainder

In [51]:

```
a = 7
b = 2
print('addition: ',a+b)
print('subtraction: ',a-b)
print('multiplication: ',a*b)
print('division: ',a/b)
print('exponentiation: ',a**b)
print('floor division: ',a//b)
print('remainder: ',a%b)
```

```
addition:  9
subtraction:  5
multiplication:  14
division:  3.5
exponentiation:  49
floor division:  3
remainder:  1
```

## Syntax notes

Python supports a short for syntax when you want to do a calculation on a variable and replace the value of the variable.

In [52]:

```
a = 8
a = a+5
print(a)
```

```
13
```

In [53]:

```
a = 8
a += 5
print(a)
```

```
13
```

So you can write more efficient versions using
+= add and replace
-= subtract and replace
*= multiply and replace
/= divide and replace
**= raise to a power and replace

## Variable Names

### The rules

- Variable names are always on the left of the =, values or expressions are always on the right.
- Variable names are case sensitive, e.g. c and C are different variables.
- Variable names must start with letters, but then can include numbers, e.g., A1, b2, C3.
- Variable names cannot include special characters (like &, *, #, etc).
- Variable names can include underscores to improve readability, e.g., A_1, b_2, C_3.

## What are good variable names?

- In general, the more explicit and self-explanatory the variable names the better.
- A program usually expresses an idea, and the variable names should make that idea easy to understand by being explicit.
- A program should read like a story!
- Sometimes, shorter variables make sense, where the short hand is widely known and recognized.

In [54]:

```
#%%Einstein in words
mass = 10 # kg
speed_of_light = 3e+08 #m/s
                      #note the use of scientific notation
Energy = mass*speed_of_light**2  #Joules
print('Energy is', Energy, 'Joules')
```

```
Energy is 9e+17 Joules
```

In [55]:

```
#Einstein in widely known variable names
m = 10 # kg
c = 3e+08 #m/s
        #note the use of scientific notation
E= m*c**2  #Joules
print('Energy is', E, 'Joules')
```

```
Energy is 9e+17 Joules
```

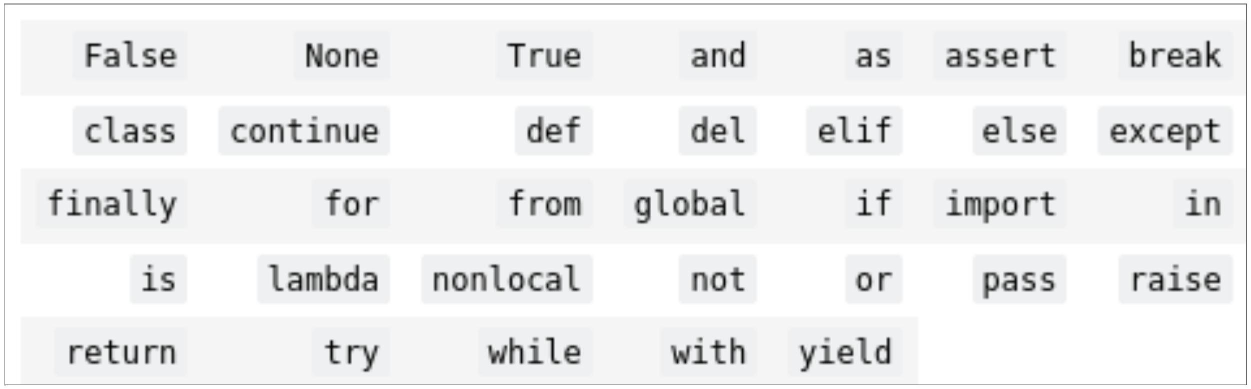
In [56]:

```
#Einstein in generic variables
x = 10 # This is the mass of the object in kg
y = 3e+08 #This is the speed of light in m/s
        #note the use of scientific notation
z = x*y**2 #This is the energy in Joules
print('Energy is', z, 'Joules')
```

```
Energy is 9e+17 Joules
```

## Reserved Words

- There are 33 words that are **never** allowed to be used as variable names



| | | | | | | |
|---|---|---|---|---|---|---|
| False | None | True | and | as | assert | break |
| class | continue | def | del | elif | else | except |
| finally | for | from | global | if | import | in |
| is | lambda | nonlocal | not | or | pass | raise |
| return | try | while | with | yield | | |

- If you try to use them, python will return an error.
- In addition, these 33 words have critical roles in python syntax, and thus are colored differently.

## Utilities

- The are some built in utilities in python that can be helpful in working with python variables.
- `type`, reports the type of variable
- `int`, converts a variable into an integer
- `float`, converts a variable into a floating point variable

In [57]:

```
#type will tell me about the type of variable
print(type(my_text_label))
print(my_text_label,'is a',type(my_text_label))
print(my_computed_variable,'is a',type(my_computed_variable))
print(my_variable,'is a',type(my_variable))
```

```
<class 'str'>
Example is a <class 'str'>
0.8 is a <class 'float'>
1 is a <class 'int'>
```

In [58]:

```
#I can convert floating point numbers into integers or integers into floating point numbers
float_variable = float(my_variable)
print(float_variable)
int_variable = int(float_variable)
print(int_variable)
```

```
1.0
1
```

In [59]:

```
#int is not the same thing as rounding a variable.
print(my_computed_variable)
int(my_computed_variable)
```

```
0.8
```

Out[59]:

```
0
```

## Working with Strings

1. Strings have additional characteristics, which for scientists are mostly useful for handling file names and text labels for data and for making graphs.
2. Most importantly, string can have variable lengths, which you can get from the utility `len`

```
In [60]:
```

```
print(my_text_label)
len(my_text_label)
```

```
Example
```

```
Out[60]:
```

```
7
```

- Unlike numerical variables, text cannot be added, however it can be concatenated

In [61]:

```
no = '#1'
thisexample = my_text_label+no
print(thisexample)
space = ' '
prettierexample = my_text_label+space+no
print('I like this better')
print(prettierexample)
```

```
Example#1
I like this better
Example #1
```

- Numeric variables can be converted to string using `str`

In [62]:

```
string57 = str(57)
print(string57)
type(string57)
```

```
57
```

Out[62]:

```
str
```

## Lists

- A list is a collection of values in a single variable.

In [63]:

```
my_list = [1,9,6,7] # I made a list here with 4 numeric entries.  Note the use of square brackets [] and commas ,
                    # to separate the elements of the list.
print(my_list)
```

```
[1, 9, 6, 7]
```

- Examine my_list in the Variables pane.
- The list is of shape 4, reflecting the number of elements, and of type list.
- The list is a flexible data type designed as a **container** for holding items together
- But its not a particularly useful data type for mathematical operations.

## Can we do math with lists?

In [64]:

```
my_list_2 = [4,7,8,9]
print(my_list_2)
add_em_up = my_list + my_list_2
print(add_em_up)
```

```
[4, 7, 8, 9]
[1, 9, 6, 7, 4, 7, 8, 9]
```

In [65]:

```
my_list_3 = my_list+3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[65], line 1
----> 1 my_list_3 = my_list+3

TypeError: can only concatenate list (not "int") to list
```

- So that went badly, because a list doesnt support mathematical operations.

The strength of lists is the flexibility to hold together different data types that are associated with each other.

In [66]:

```
#I can mix together all the data types I know about in a list
my_crazy_list = ['Kawhi','Leonard','June',29,1991,79.0]
```

I created a list all associated with the best *two-way* NBA player, including

- *strings* first name, last name, and date of birth,
- *integers* month and year of birth
- *floating point* height

## List methods: `append`

- The flexibility of lists is their strength. It is also their weakness as the flexibility also gives greater scope for making mistakes in data handling.
- `list` is a **class** with **methods**.
- Here I show how we **append** elements to a list
- The syntax here is different from our usual syntax
- `list_name.append(variable)`
- notice that instead of setting the list or variable equal to something, I used a `.` to separate the list name from the instruction to append.

In [67]:

```
my_Kawhi_list = ['Kawhi','Leonard','June',29,1991,79.0]
points = 23.7
rebounds = 6.1
assists = 3.6
steals = 1.6
blocks = 0.9
#lets add these numbers to the list
my_Kawhi_list.append(points)
my_Kawhi_list.append(rebounds)
my_Kawhi_list.append(assists)
my_Kawhi_list.append(steals)
my_Kawhi_list.append(blocks)
#
print(my_Kawhi_list)
```

```
['Kawhi', 'Leonard', 'June', 29, 1991, 79.0, 23.7, 6.1, 3.6, 1.6, 0.9]
```

In [68]:

```
#In this example I am going to start with an empty list
Kawhi_shooting = list()
Kawhi_shooting.append(52.5)
Kawhi_shooting.append(41.7)
Kawhi_shooting.append(88.5)
print(Kawhi_shooting)
#I could also make an empty list like this
Kawhi_availability = []
Kawhi_availability.append(68)
Kawhi_availability.append(34.3)
print(Kawhi_availability)
```

```
[52.5, 41.7, 88.5]
[68, 34.3]
```

## Indexing into lists

- Now suppose i wanted to get to his points per game for some data analysis (23.7). .
- I want to figure out a way to do it so that for every player whose data is organized in the same order
- I can extract the points per game from the list.

In [69]:

```
print(my_Kawhi_list)
```

```
['Kawhi', 'Leonard', 'June', 29, 1991, 79.0, 23.7, 6.1, 3.6, 1.6, 0.9]
```

- We can see that points is the 7th entry in the list.

In [70]:

```
ppg = my_Kawhi_list[7]
print(ppg)
```

```
6.1
```

- That didnt work. In fact, it returned the 8th element of the list, which is rebounds.
- Computers count from zero, while human beings count from 1.
- If we start the count with 'Kawhi' as item 0, we realize the **index** for 26.6 is 6 and not 7

In [71]:

```
ppg = my_Kawhi_list[6]
print(ppg)
```

```
23.7
```

## Lists do not have data types, but the values contained within them retain their types.

- Take a look at the Variables pane, or print the type of ppg.
- Notice that the variable ppg is of type float.
- Inside a list values cannot have a type, but once I remove a variable from the list it has a type.

In [72]:

```
print(type(ppg))
```

```
<class 'float'>
```

## Index to a range of values.

- Suppose I want to recover Kawhi's points, rebounds, assists - the classic box score stats.
- Since points is item 6 counting from 0, item 7 is rebounds and item 8 is assists (see above to verify)

In [73]:

```
boxscore = my_Kawhi_list[6:8]
print(boxscore)
```

```
[23.7, 6.1]
```

- That didnt work either.
- Since we need the next item, lets guess we should add one more and go from 6:9

In [74]:

```
boxscore = my_Kawhi_list[6:9]
print(boxscore)
```

```
[23.7, 6.1, 3.6]
```

Indexing in Python is always **inclusive** of the first element and **exclusive** of the last element.

## Indexing to the end of the list

- Suppose we want to get a list of all numerical values of basketball statistics.
- The 7th element of the array is ppg, up to the end of the list

In [83]:

```
gamestats = my_Kawhi_list[6:] #notice i didnt put an upper bound on this.
print(gamestats)
```

```
[23.7, 6.1, 3.6, 1.6, 0.9]
```

## Indexing **backwards** from the end of a List!

- When indexing into list, we sometimes want to get the last element, or a certain number of elements from the end of the list.
- Of course if you know how long the array is in advance, you can easily solve this.
- Python supports indexing from the end of an array using negative indexes.

In [84]:

```
blocks = my_Kawhi_list[-2]
print(blocks)
blocks_steals = my_Kawhi_list[-2:]
print(blocks_steals)
```

```
1.6
[1.6, 0.9]
```

## Merging Lists

1. We can merge two lists in the same way we handled strings, with a simple +
2. Lists and strings do not do math addition, instead + means concatenation.

In [85]:

```
my_kawhi_list = ['Kawhi','Leonard']
points = 23.7
rebounds = 6.1
assists = 3.6
steals = 1.6
blocks = 0.9
```

In [86]:

```
kawhi_season_stats = [points,rebounds,assists,steals,blocks]
merged_kawhi_list = my_kawhi_list + kawhi_season_stats   # + for lists does not do ADDITION.
print(merged_kawhi_list)
merged_kawhi_list.append(Kawhi_shooting)
print(merged_kawhi_list)
```

```
['Kawhi', 'Leonard', 23.7, 6.1, 3.6, 1.6, 0.9]
['Kawhi', 'Leonard', 23.7, 6.1, 3.6, 1.6, 0.9, [52.5, 41.7, 88.5]]
```

You can append an individual variable to a list. If that variable is a list, it becomes a single item in the new list. To merge lists always use '+'

**Strengths of Lists:**

- Flexible data type that allows you to keep together different type of informtion.
- Can be easily indexed, and manipulated using `append` and other methods

**Weaknesses of Lists:**

- Because it can contain many data types, it cannot be used for math.
- Only the person who creates the list knows how to index into it.

## Dictionaries

A bit more structured, and more cumbersome, but far more informative.
Instead of an index you use a `key` to tell you where something is placed in the dictionary.
Each `key` is associated with a `value` which can be any type of variable, including strings, numerical values, lists, or even another dictionary

In [87]:

```
my_Kawhi_dict = dict()
my_Kawhi_dict['First Name'] = 'Kawhi'
my_Kawhi_dict['Last Name'] = 'Leonard'
my_Kawhi_dict['Date of Birth'] = ['June', 29, 1991]
my_Kawhi_dict['points'] = 23.7
my_Kawhi_dict['shooting'] = Kawhi_shooting
```

In [88]:

```
print(my_Kawhi_dict['points'])
shooting = my_Kawhi_dict['shooting']
print(shooting)
```

```
23.7
[52.5, 41.7, 88.5]
```

Now suppose you made a dictionary and you forgot the keys

In [89]:

```
my_Kawhi_dict.keys()
```

Out[89]:

```
dict_keys(['First Name', 'Last Name', 'Date of Birth', 'points', 'shooting'])
```

Another way to build a dictionary (I am not a fan) is a keyword:value pair

In [90]:

```
Kawhi_stats = {'points':points, 'rebounds':rebounds, 'assists':assists, 'shooting':shooting}
```