# Lab01A

Revision Part A

# Major Topics

- Data Types & Type Conversion
- Calculation
- String manipulation
- Loops
- Nested Loops

# Data Types

- In Python, all data has an associated data "Type".
- You can find the "Type" of any piece of data by using the type() function:

```
type( "Hi!") produces <class 'str'>
type( True ) produces <class 'bool'>
type( 5) produces <class 'int'>
type(5.0) produces <class 'float'>
x = 5,
type(x) produces <class 'tuple'>
```

- Type Conversion
  - Functions exist which will take data in one type and return data in another type.
    - int() - Converts compatible data into an integer. This function will truncate floating point numbers
    - float() - Converts compatible data into a float.
    - str() - Converts compatible data into a string.

# Variables

- Variables are names that can point to data.
  - They are useful for saving intermediate results and keeping data organized.
- Every variable in Python is created when it's assigned ('=') a value
  - name = "Bob"
  - Don't confuse the assignment operator (single equal sign, =) with the Equality-Test operator (double equal sign, ==)
- Variable name rules
  - No spaces
  - No keywords (words that already have special meaning)
  - Must start with a letter
  - camelCase or underscores_for_spaces

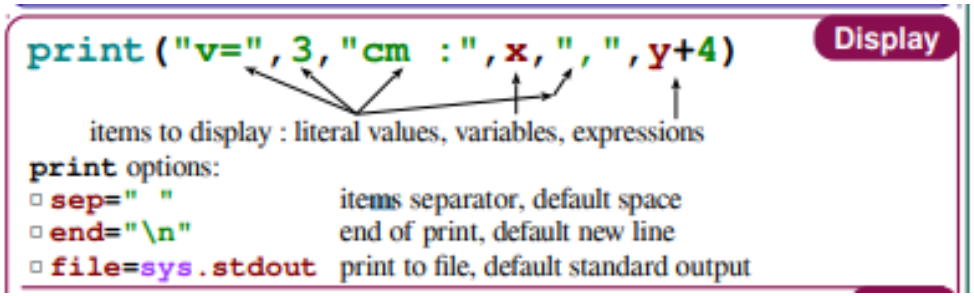
# Operators & Expressions

- Python has many operators. Some examples are:
  - +, -, *, /, //, %, >, <, ==, >=, <=
  - Operators perform an action on one or more operands. Some operators accept operands before and after themselves:
- An expression is any set of values and operators that will produce a new value when evaluated. Here are some examples, along with the new value they produce when evaluated:

```
5 + 10                    produces 15
'Hi' + " " + 'Jay!'       produces "Hi Jay!"
10 > 5                    produces True
10 / 3.5                  produces 2.8571428571
10 // 3                  produces 3
10 % 3                    produces 1
```

- Operator overloading
  - "Hi" + "Jay" produces "HiJay"
  - "Hi Jay" * 3 produces "Hi JayHi JayHiJay"

# Built-in functions



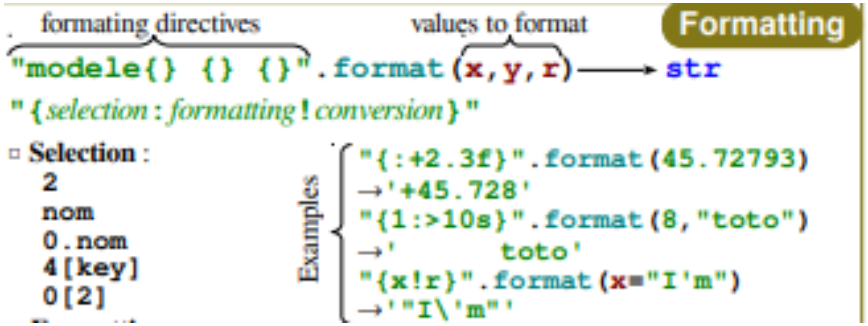

- Print
  - Python allows us to print a collection of things, separated by commas. It will insert spaces automatically:
    - `print("The sum of", num1, "and", num2, "is", num3)`
  - Alternately, you can use 'concatenation' (i.e. "+") to join things together

```
print( "The sum of " + num1 + " and " + num2 " is " + num3)
```

    - We MUST explicitly insert spaces
  - Or, use a format string

```
print("The area is {:.2f}".format(area))
```

    - print the result in 2 decimal places



# Getting input from the User

- Built-in function:– input
  - variable_name = input(prompt)
  - Displays prompt and then gets a value from the keyboard.
  - This value will be stored in variable_name a string
  - Typecast to an int or float if needed
    - number = int(input(prompt))
    - value = float(input(prompt))
- Other functions:

**Generic Operations on Containers**

`len(c)` → items count
`min(c)` `max(c)` `sum(c)`
`sorted(c)` → `list` sorted *copy*
`val in c` → boolean, membership operator `in` (absence `not in`)
`enumerate(c)` → *iterator* on (index, value)
`zip(c1,c2…)` → *iterator* on tuples containing $c_i$ items at same index
`all(c)` → `True` if **all** `c` items evaluated to true, else `False`
`any(c)` → `True` if **at least one** item of `c` evaluated true, else `False`

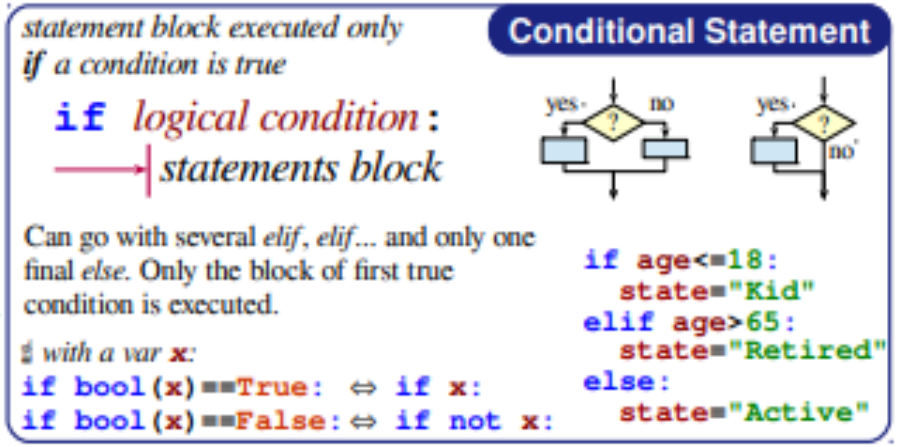
*Note: For dictionaries and sets, these operations use* ***keys****.*

*Specific to* ***ordered sequences containers*** *(lists, tuples, strings, bytes…)*
`reversed(c)` → *inversed iterator* `c*5` → duplicate `c+c2` → concatenate
`c.index(val)` → *position* `c.count(val)` → *events count*

```
import copy
copy.copy(c) → shallow copy of container
copy.deepcopy(c) → deep copy of container
```

# Making Decisions

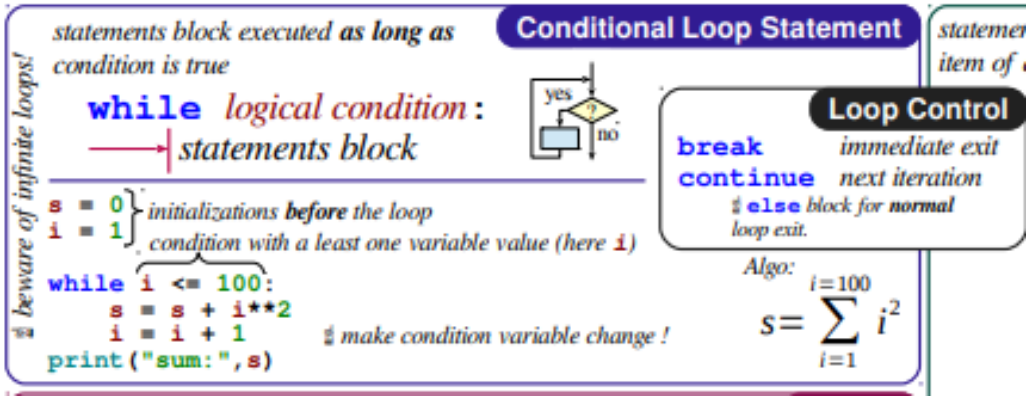
- The IF statement allows you to conditionally execute a block of code.
  - The indented block of code following an if statement is executed if the boolean expression is true, otherwise it is skipped




  - If you have two mutually exclusive choices, and want to guarantee that only one of them is executed, you can use an IF/ELSE statement. The ELSE statement adds a second block of code that is executed if the boolean expression is false.

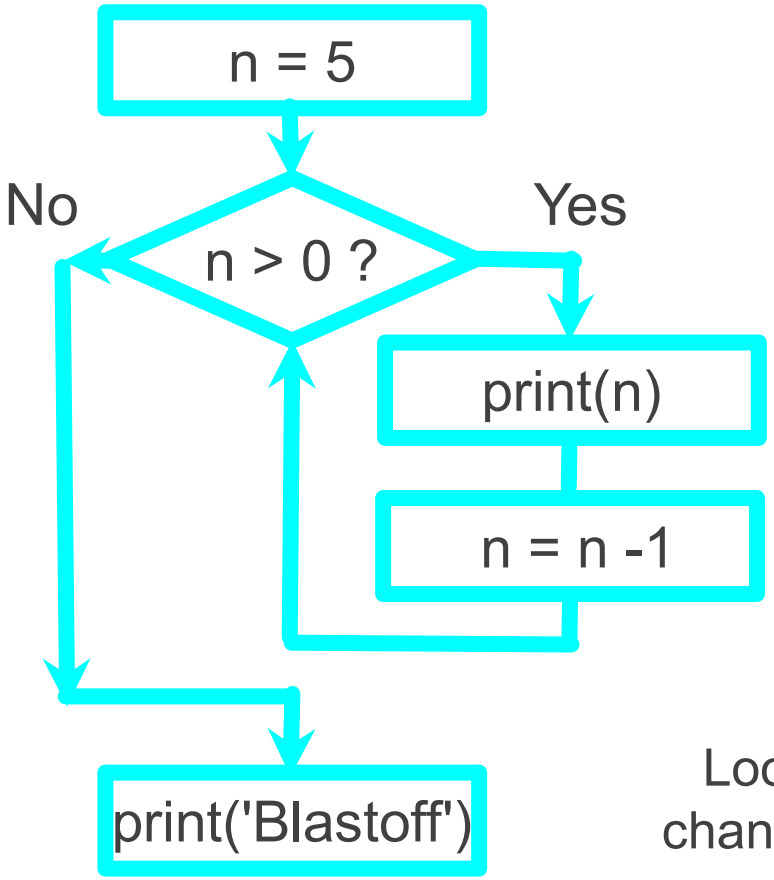
# Repetition

- The while loop repeats a block of code until a boolean expression is no longer true.




- Steps:
  - Initialize the loop counter (to zero)
  - Test the loop counter in the boolean expression (is it smaller than 100, if yes, keep looping)
  - Increment the loop counter (add one to it) every time we go through the loop
- If we miss any of the three, the loop will NEVER stop!

# Repeated Steps




Program:

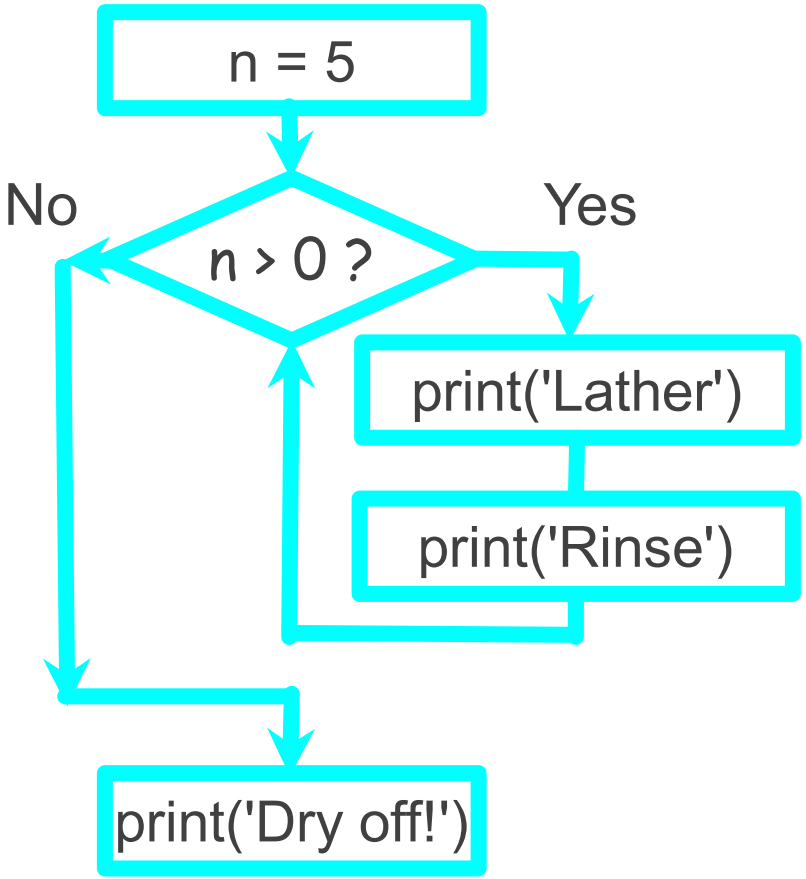
```
n = 5
while n > 0 :
    print(n)
    n = n - 1
print('Blastoff!')
print(n)
```

Output:

5
4
3
2
1
Blastoff!
0

Loops (repeated steps) have iteration variables that change each time through a loop. Often these iteration variables go through a sequence of numbers.

# An Infinite Loop



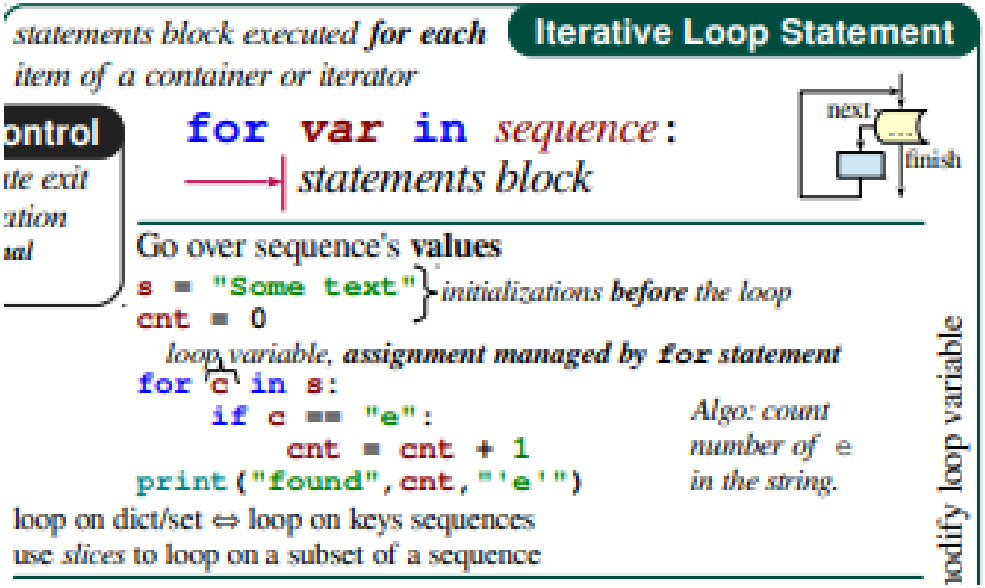

```
n = 5
while n > 0 :
    print('Lather')
    print('Rinse')
print('Dry off!')
```

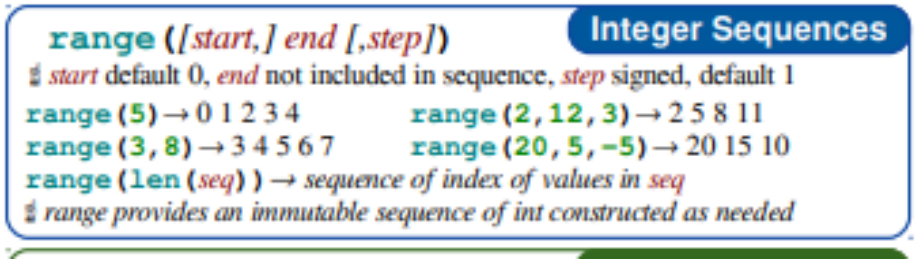
What is wrong with this loop?

# For loop

- We can use for statement for looping over
  - A list
  - A string
  - Keys from a dictionary
  - Lines within a file

- These are called iterable objects



# Slicing

**Sequence Containers Indexing**

*for lists, tuples, strings, bytes…*

| | | | | | |
|---|---|---|---|---|---|
| *negative index* | -5 | -4 | -3 | -2 | -1 |
| *positive index* | 0 | 1 | 2 | 3 | 4 |
| `lst=[10, 20, 30, 40, 50]` | | | | | |
| *positive slice* | 0 | 1 | 2 | 3 | 4 | 5 |
| *negative slice* | -5 | -4 | -3 | -2 | -1 | |

**Items count**

`len(lst)→5`

**index from 0**

(here from 0 to 4)

Individual access to **items** via `lst[`*index*`]`

`lst[0]→10` ⇒ *first one* `lst[1]→20`

`lst[-1]→50` ⇒ *last one* `lst[-2]→40`

*On mutable sequences (`list`), remove with* `del lst[3]` *and modify with assignment* `lst[4]=25`

Access to **sub-sequences** via `lst[`*start slice*`:`*end slice*`:`*step*`]`

`lst[:-1]→[10,20,30,40]` `lst[::-1]→[50,40,30,20,10]` `lst[1:3]→[20,30]` `lst[:3]→[10,20,30]`

`lst[1:-1]→[20,30,40]` `lst[::-2]→[50,30,10]` `lst[-3:-1]→[30,40]` `lst[3:]→[40,50]`

`lst[::2]→[10,30,50]` `lst[:]→[10,20,30,40,50]` *shallow copy of sequence*

*Missing slice indication → from start / up to end.*

*On mutable sequences (`list`), remove with* `del lst[3:5]` *and modify with assignment* `lst[1:4]=[15,25]`

# Help for the Lab 01A

- Q3: the python swapcase() method may be helpful
- Q4: sorting the letters in each string is a good way to check if two strings are anagrams. (Using the sorted() function)
- Q6: Since we don't know how many iterations we need to do, a **while** loop might be helpful here
- Q8: if you need to repeat a character n times you can use the * operator (e.g. **"x" * 5** would produce **"xxxxx"**)
- Q9: you can create a tuple with the tuple() function (e.g. **tuple([1, 2, 3])** would create the tuple **(1, 2, 3)**))
- Q10: If you want to loop over the keys in a dictionary use the keys() method.
  (e.g. **for key in sorted(my_dictionary.keys()): ...** will loop through all the keys in sorted order)