

## Today's Topics:

- Numpy

**Lee wilkins**

Office : 6C.9

Contact: MIO

# INTRODUCTION TO COMPUTER PROGRAMMING IN ENGINEERING AND SCIENCE

## Comp Sci assignments (All tests and assignments occur Wednesdays in our lab block)

2

Test 1 week 7 (15%) Wednesday March 5)

Assignment 2 week 8 (10%)

Assignment 3 week 11 (10%)

Test 2 week 13 (15%)

## Physics assignments

Assignments (4 x 2%) 8% Date communicated by the Physics teacher

Project 1: Solving differential equations 10% Week 11

Project 2: Applying programming in science 22% Week 15

# GRADE BREAKDOWN REVIEW

# STRINGS

String operators also follow BEDMAS/PEDMAS

```
greeting = "hello"  
tone = "!"  
print(greeting + tone*3)  
# hello!!!  
print((greeting+tone)*3)  
# hello!hello!hello!
```

# STRING OPERATORS

Indexing is counting through the characters (letters, digits, etc) in a string

Python uses zero-based indexing

`string[0]` gives us the first character of the string

```
greeting = "hello"  
print(greeting[0])
```

Negative index numbers count backwards from the end of the string

`string[-1]` gives us the last character of the string

```
greeting = "hello"  
print(greeting[-1])
```

# STRING INDEXING

We can use slicing to fetch a **substring** - a part of a string

`some_string[2:6]`

This will give us the substring **from the 3rd character to the 6th**

```
greeting = "greeting"  
print(greeting[2:6])  
>> eeti
```

G	R	E	E	T	I	N	G
0	1	2	3	4	5	6	7

Remember, Python uses **zero-indexing**

The 7th character (index 6) is not included

You may find it helpful to imagine the indexes as counting *between* the characters

# STRING SLICING

```
greeting = "greeting"  
print(greeting[6])  
# >> n just 6  
print(greeting[:6])  
# >> greeti before 6  
print(greeting[6:])  
# >> ng 6 and after
```

G	R	E	E	T	I	N	G
0	1	2	3	4	5	6	7

# STRING SLICING

**Interpolation** lets us use variables inside of a string

In Python, this is called **string formatting** or an **f-string**

```
greeting = f'Hello {name}, it is {day_of_week} today!'
```

**f** before the **starting quote mark**

**Variable names** in curly braces { }

This is new syntax as of Python 3.6

```
name = "lee"  
day_of_week = 5  
greeting = f'Hello {name}, it is {day_of_week} today!'  
print(greeting)
```

# STRING INTERPOLATION



Sometimes in a string, we want to use a special character that Python might misinterpret, like ' or " or { }

These characters need to be **escaped** so Python understands they're part of the string

In Python, we can escape a character by putting a \ in front of it

e.g.: `print('It\'s raining outside.')`

This means the \ character also needs to be escaped if it's part of the string (\\)

[https://www.w3schools.com/python/gloss\\_python\\_escape\\_characters.asp](https://www.w3schools.com/python/gloss_python_escape_characters.asp) Here is a list of them

`\n` is for a new line

# A NOTE ON SPECIAL CHARACTERS...

A **method** is a function that acts on a specific object

Python uses dot notation for methods

Some methods **return** (give back to us) a property of the object

```
print(greeting.count("e")) # counts the number of es, returns 2
print(len(greeting)) # tells us the length of the string (8),
#length worts differently (sorry)
```

Some methods return a modified version of the object

```
print(greeting.upper())
# prints GREETING
print(greeting.lower())
# prints greeting
https://www.w3schools.com/python/python\_ref\_string.asp find more here
```

# STRING METHODS

```
#here I am printing my string, note the case.  
my_test_string = "hEllo thIS is a tEst"  
print(my_test_string)  
# here I just print the variables, but I am not storing them anywhere  
# the methods return the value inside the print function  
print(my_test_string.upper())  
print(my_test_string.lower())  
# but my variable still has its mixed case  
print(my_test_string)  
#here I am storing the upper and lower case strings in new variables,  
# but not printing them  
my_test_string_uppercase = my_test_string.upper()  
my_test_string_lowercase = my_test_string.lower()  
# now i am printing my new variables, with their upper and lower case contents  
print(my_test_string_uppercase)  
print(my_test_string_lowercase)
```

# STRING METHODS, STORING AND PRINTING

This can be useful for cleaning up strings that the user inputs OR for clearing up data. For example, data might have extra spaces, capital letters or characters that mean your strings do not match.

"mystring" will not equal "mystring " or "myString"

.lower() can be used to make all letters lower case

.strip() will remove white space (there is .lstrip() and .rstrip() for right and left)

```
my_answer = input("what is the question?")
if "yes" in my_answer.lower():
    print("its there")
else:
    print("its not there")
```

# STRING METHODS

```
my_Var = "this is a test " # this one contains white space
if my_Var == "this is a test": #therefor it does not equal this string
    print("it matches") # this condition will never be met
```

```
my_Var = my_Var.strip() # I can use strip to remove white space
if my_Var == "this is a test": # now it equals the string
    print("it matches")
else:
    print("not yet")
```

```
my_Var = my_Var.replace(" ", "") # i can remove all white space by using replace
print (my_Var)
```

# STRING METHODS

**Lists** are a way of storing many variables (in this case, strings). You can convert a string to a list.

We will talk more about lists later, but you can look ahead at the list methods [https://www.w3schools.com/python/python\\_ref\\_list.asp](https://www.w3schools.com/python/python_ref_list.asp)

You can access list items like this

`my_list[0]` first item in list

`my_list[1]` second item in list

# STRINGS TO LISTS

```

my_list = "a dog, a cat, a donkey"
print(my_list[0])
# prints a
print(type(my_list))
# prints <class 'str'>
my_list = my_list.split(",")
print(my_list[0])
# prints a dog
print(type(my_list))
# prints <class 'list'>

```

**Split** can be used to break a string up into a **list**. The split method takes one parameter, it is **whatever delineator you want to split your string by**, surrounded by **quotes**

In this example, we split the string by **" , "**

But we could split a string by spaces, | , or any other word or series of words

```

my_list = "dog cat donkey"
my_list = my_list.split(" ")
>> my_list[0] > "dog"
my_list = "dog|cat|donkey"
my_list = my_list.split("|")
>> my_list[0] > "dog"

```

# STRING METHODS

**MATH**



```
import math
```

This gives access to all kinds of things to help you do more advanced calculations. You can see all of them here.

You need to put this at the top of your document or above all of the math module methods you use.

[https://www.w3schools.com/python/module\\_math.asp](https://www.w3schools.com/python/module_math.asp)

# MATH MODULE

```
print(math.pi) # pi  
print(math.e) # Euler's number  
print(math.tau) # tau
```

By importing math we have access to a number of different **methods** and **constants**. These are constants that don't have () because they are not performing tasks, they are representing numbers.

# CONSTANTS

```
my_num = math.sqrt(4)  
print(my_num)
```

You can pass numbers **through** a **method** as a **parameter** (or param). In this example, we are passing **4** through the square root function (math.sqrt())

You can always look up how it works

[https://www.w3schools.com/python/ref\\_math\\_sqrt.asp](https://www.w3schools.com/python/ref_math_sqrt.asp)

# METHODS

You can convert degrees and radians by passing them **THROUGH** the appropriate methods. You can pass a method a **parameter** by putting it in the round brackets like this:

```
print(math.degrees(1))  
# this takes radians, prints degrees  
print(math.radians(90))  
# this takes degrees, prints radians
```

You can also pass **variables** through a function

```
my_angle_degrees = 90  
my_angle_radians = math.radians(my_angle_degrees)  
print(f'{my_angle_degrees} degrees is {my_angle_radians} radians')  
  
another_angle_radians = 1.1  
another_angle_degrees = math.degrees(another_angle_radians)  
print(f'{another_angle_degrees} degrees is {another_angle_radians} radians')
```

# USING MATH.

```
my_num = math.pow(2,2)  
print(my_num)
```

Some methods need **multiple parameters**, such as pow. This needs to know the **base** and the **exponent**

[https://www.w3schools.com/python/  
ref\\_math\\_pow.asp](https://www.w3schools.com/python/ref_math_pow.asp)

# USING MATH.

You can pass methods through other methods. Like this:

```
myNum = 2.0
```

```
print(math.sqrt(math.pow(myNum, 2)))  
> 2.0
```

```
print(math.log(math.exp(myNum)))  
> 2.0
```

```
print(math.log10(math.pow(10, myNum)))  
> 2.0
```

# USING MATH.

```
opposite_side = 5  
adjacent_side = 3
```

These two lines are the same

```
hypotenuse_side = opposite_side**2 + adjacent_side**2
```

This is the same line using math.pow

```
hypotenuse_side = math.pow(opposite_side, 2) + math.pow(adjacent_side, 2)
```

I can apply the square root function after

```
hypotenuse_side = math.sqrt(hypotenuse_side)
```

You can also do it all in one expression like this

```
hypotenuse_side = math.sqrt(math.pow(opposite_side, 2) + math.pow(adjacent_side, 2))
```

```
print(f'The triangle hypotenuse is {hypotenuse_side}')
```

Remember, it is not faster to do all the calculations in one line. Don't hesitate to spread it out and use multiple lines. Make it easy to read!

# USING MATH.

Calculate the adjacent angle with the two sides, your result is in **radians**.

```
angle_adjacent = math.atan(opposite_side/adjacent_side) Returns the value in radians
```

Convert it to **degrees** like this

```
angle_adjacent = math.degrees(angle_adjacent)
```

Or all in one expression where degrees holds the entire expression

```
angle_adjacent = math.degrees(math.atan(opposite_side/adjacent_side))
```

# USING MATH.



IF

Booleans are data types that are either True or False.

They are similar to int, float, string, in that they are a type of information@

Note the capital T and F!

```
my_boolean = True;
```

```
print(my_boolean)
```

```
if my_boolean: my_boolean = False
```

```
print(my_boolean)
```

# BOOLEANS

```
myNum = int(input("Enter a number: "))  
myNum2 = int(input("Enter another number: "))
```

```
my_boolean = myNum > myNum2
```

Boolean values can be assigned as the result of a comparison like this. In this example, my\_boolean will be true or false. We can print it and see based on the answers.

# BOOLEANS

If statements check to see if your statement is true. If the result of the comparison<sup>28</sup> is true, then the commands in the if statement are executed.

```
my_var = True
if my_var == True:
    print("its true")
```

```
my_int = 4
if my_int > 2:
    print(f'{my_int} is greater than 2')
```

```
if "hello" in my_string:
    print("Hello is in the string")
```

# IF

I can use an else command if my statement is not true. So, if anything else is true, it will be executed.

```
my_string = " This is an example, it doesn't contain the secret word."
if "hello" in my_string:
    print("Hello is in the string")
else: # If anything else is true, else will be executed.
    print("The string doesn't contain hello")
```

# ELSE

```
my_string = " This is an example, it doesn't contain the secret word."
```

```
if "hello" in my_string:  
    print("I am looking for hello in the string, and I found it")  
else:  
    print("I am looking for hello in the string, and I didn't find it")
```

```
if "hello" not in my_string:  
    print("I am looking for a string that DOESNT contain hello, and I found  
hello")  
else:  
    print("I am looking for a string that DOESNT contain hello, and I didn't  
find hello ")
```

I can use the keyword not in in order to check if something is NOT IN a string

# NOT

We can use an AND or OR chain in our if statements to compare two sets.

In this case, we are comparing the same two statements but with logical and or OR

```
my_bool = False
my_int = 4
if my_int > 2 and my_bool == True :
    print("Both ARE true")
else:
    print("Both are NOT") # This one will run

if my_int > 2 or my_bool == True :
    print("only one has to be true for this to happen") #this one will run i
else:
    print("Neither are true")
```

# AND, OR

We can chain multiple if statements with an elif statement. Each elif must have a condition. Else statements do not have conditions.

These will be **executed in order** (top to bottom), but **only one will ever run**.

```
temp = 40
if temp >= 30:
    print("its real warm out")
elif temp >= 20:
    print("its a bit warm")
elif temp >= 10:
    print('getting cold!')
else:
    print("so cold!!!")
```

# ELIF



If we don't use an else statement, all if statements will all be tested.

In this example, all of the statements are true, and therefor all get executed! For this example, we don't want that because we only want one to run, so this won't work.

```
temp = 40
if temp >= 30:
    print("its real warm out")
if temp >= 20:
    print("its a bit warm")
if temp >= 10:
    print('getting cold!')
```

# IF WITHOUT THE ELSE

However, in this example we **WANT** to check each example to check if the person has too many animals. We want each if statement to run.

```
def animal_limits():  
    number_of_birds = 10  
    number_of_cats = 30  
    number_of_hamsters = 20  
    animal_threshold = 15  
    if number_of_birds > animal_threshold:  
        print("you have too many birds")  
    if number_of_cats > animal_threshold:  
        print("you have too many cats")  
    if number_of_hamsters > animal_threshold:  
        print("you have too many hamsters")
```

```
animal_limits()
```

# IF WITHOUT THE ELSE

We can make this into a function called `weather_function()` that asks

```
def weather_function():  
    temp = int(input("What is the temp?"))  
    if temp >= 30:  
        print("its real warm out")  
    elif temp >= 20:  
        print("its a bit warm")  
    elif temp >= 10:  
        print('getting cold!')  
    else:  
        print("so cold!!!")
```

Now call the function:

```
weather_function()
```

# MAKING IT A FUNCTION

```
def student_cookies():  
    num_students = int(input("How many students?"))  
    num_cookies = int(input("how many cookies?"))  
    students_are_good = input("Are the students good? y/n")  
    if num_cookies >= num_students:  
        print("there is enough cookies")  
        if "y" in students_are_good.lower():  
            print("and they are good, so they get cookies")  
        else:  
            print("but the students aren\'t good, so no cookies!")  
    else:  
        print("there are not enough cookies :(")  
  
student_cookies()
```

I can use an if statement inside of an if statement in order to make complex decisions. **This is where its a good time to use pseudo code or flow charts to understand your problem!**

# NESTED IF

```
def student_cookies():  
    try:  
        num_students = int(input("How many students?"))  
        num_cookies = int(input("how many cookies?"))  
        students_are_good = input("Are the students good? y/n")  
        if num_cookies >= num_students:  
            print("there is enough cookies")  
            if "y" in students_are_good.lower():  
                print("and they are good, so they get cookies")  
            else:  
                print("but the students aren't good, so no cookies!")  
        else:  
            print("there are not enough cookies :(")  
    except ValueError:  
        print("gotta be the right data type")  
  
student_cookies()
```

More next week: we can catch errors and perform tasks based on them. In this example, if the user inputs the wrong values, we get an error.

# WITH VALUE ERRORS

# FUNCTIONS

def means we  
are defining the  
function

This is your  
function name

Note the syntax,  
you always need  
( ) so you know its  
a function

```
def my_function():  
    print("this is my function")
```

Function content is indented

Your function ends, here, but it wont be called until you do this:

```
my_function()
```

This is where your  
function actually runs.  
This is called **calling the  
function**.

# FUNCTION SYNTAX

I can pass a parameter through my function. It will be used by this name only within the scope of the function.

Whatever I pass through the function will be placed here

```
def my_function_param(my_string):  
    print(my_string)  
  
my_function_param("this is a string")  
my_function_param("I can print anything here")  
my_function_param("This function is so useful!")
```


# PASSING PARAMETERS

We can call the same function to produce different results



You can create variables inside your function. They only exist inside the function and not in the rest of your program

You can pass multiple parameters through your function



```
def calculate_square_area(width, height):  
    area = width * height  
    print(f'The area of a square with the width of {width} inches and the height of {height} inches is {area} inches^2')
```

```
calculate_square_area(2,4)
```

```
# The area of a square with the width of 2 inches and the height of 4 inches is 8 inches^2
```

```
calculate_square_area(5,10)
```

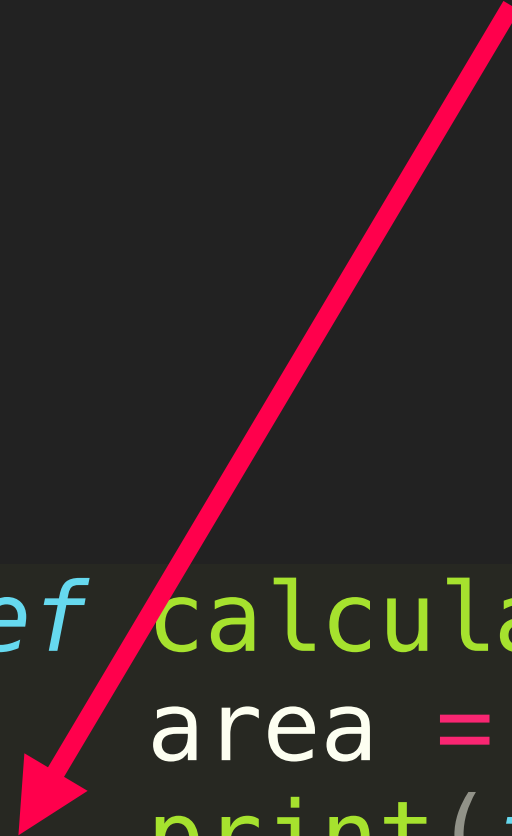
```
# The area of a square with the width of 5 inches and the height of 10 inches is 50 inches^2
```

```
calculate_square_area(100,3)
```

```
# The area of a square with the width of 100 inches and the height of 3 inches is 300 inches^2
```

# MULTIPLE PARAMETERS

The function ends when  
the indentation ends



```
def calculate_square_area(width, height):  
    area = width * height  
    print(f'The area of a square with the width of {width} inches ad the height of  
{height} inches is {area} inches^2')
```

```
print(area)
```

```
# NameError: name 'area' is not defined
```

Area only exists inside the  
function. This idea is  
called scope

# SCOPE

```
def return_calculate_square_area(width, height):  
    area = width * height  
    return(area)
```

```
print(return_calculate_square_area(2,4))
```

The word return is used to  
give back a value

```
my_area = return_calculate_square_area(2,4)
```

In this example we are  
returning the area, either  
printing it or storing it

# RETURN

# ERRORS

We can use **try / except** to find errors and handle them appropriately. There are two main types of errors we will focus on for now:

**Type Error:** This occurs when you're trying to perform an action on the wrong type of data. Ex: trying to do a mathematic operation on a string!

**Value Error:** The type is correct, but the value is out of range or incorrect. For example, doing `math.sqrt` on a negative number, or trying to convert a string that contains only letters to an int.

# TRY / EXCEPT

```
try:  
except ValueError:
```

Try / Except is a way to handle errors. In order to work best, you need to know what type of error you are expecting. If anything in try encounters an error, it will perform what is inside except.

# TRY / EXCEPT

```
import math
try:
    math.sqrt(-1)
except ValueError:
    print("cannot square neg num")
```

We can use try / except like this to check for an error and perform an action. Note syntax and indentation.

# TRY / EXCEPT

```
import math
try:
    myNum = int(input("Enter a neg number"))
    math.sqrt(myNum)
except ValueError:
    print("cannot square neg num")
    why = input("Why did you do that ?!!")
```

This raises a value error because while it is a number, it is out of range for `math.sqrt` which only takes negative numbers. This is useful here because we don't know what the user will input.

# VALUE ERROR



```
my_var = "hello"  
print(my_var + 4)  
TypeError: can only concatenate str (not "int") to str
```

This causes a type error, we can't add a str and an int! We can use an exception here to catch the type error raised.

```
try:  
    my_var = "hello"  
    print(my_var + 4)  
except TypeError:  
    print("can't add an int to a string")  
    my_var = input("Please enter a new number")
```

This raises a type error because the data type is wrong for the operation we want to perform.

# TYPE ERROR

# LISTS

Before we can really see the power of for loops, we need to talk about lists.

Lists are a way of storing many things in a single variable. You can access them like we do with string indexes, remembering 0 is the first item in a list. A list can be many o

```
my_string_list = ["apple", "oranges", "bananas"]
print(my_string_list[0]) # apples
my_int_list = [2, 3, 10]
print(my_int_list[1]) #3
my_float_list = [2.4, 502.4, 2.5]
print(my_float_list[2]) #2.5
my_list_list = [ [1,4,5], [3,5,4], [4,2,5] ]
print(my_list_list[1][1]) #5
```

# LISTS

Lists can contain multiple data types. List is the entire structure (with its own methods) and each item can be accessed and on its own. Its important to pay attention to data types if your list is like this!

```
my_mixed_list = [2.4, "502.4", 2]  
print(type(my_mixed_list)) # <class 'list'>  
print(type(my_mixed_list[1])) # <class 'str'>
```

# LISTS

Similar to strings, there are list methods. You can find them here [https://www.w3schools.com/python/python\\_ref\\_list.asp](https://www.w3schools.com/python/python_ref_list.asp)

List methods can only be performed on variables that have the data type list.

```
fruits = ['apple', 'banana', 'cherry']  
print(fruits) # ['apple', 'banana', 'cherry']  
fruits.reverse()  
print(fruits) # ['cherry', 'banana', 'apple']
```

# LISTS METHODS

Append will add a new item to the list (to the end), pop will remove at an index, insert will add at an index

```
new_fruit = input("What is another fruit?")  
fruits.append(new_fruit)  
print(fruits)
```

```
mylist = ["h", "e", "l", "p"]  
print(mylist) # ['h', 'e', 'l', 'p']  
mylist.pop(0)  
print(mylist) # ['e', 'l', 'p']
```

```
mylist.insert(0, "w")  
print(mylist) # ['w', 'e', 'l', 'p']
```

# LISTS: APPEND, POP, INSERT

We can turn a string into a list if it has a delineator (something to separate the items of a list.

55

```
# Convert a string into a list
my_string_to_convert = "apples, oranges, bananas"
print(my_string_to_convert) # apples, oranges, bananas
my_string_to_convert = my_string_to_convert.split(",") # ['apples', '
oranges', ' bananas']
print(my_string_to_convert)
```

This string is split by the comma, but it has spaces! We can handle this two ways:

- Make a string that has no spaces ("apples, oranges, bananas") or trip the white space with replace

```
my_string_to_convert = my_string_to_convert.replace(" ", "")
print(my_string_to_convert)
```

# STRING TO LIST

Delimiters can be anything! “this|is|my|string” or even “this is my string” where the delimiter is a space.

This can be useful when getting data from a larger file that you need to clean up. For example, data from a study or collection!

## STRING TO LIST



# LOOPS

For: Loops through a range of items, or for each item in a set of items

While: Loops while a condition is true (be careful!)

# LOOPS

While loops can happen until the condition is no longer **true** OR there is a **break**.

Here, we are asking a question until

```
secret_number = 42
while True:
    user_guess = int(input("Guess a number between 1 and 100: "))
    if user_guess == secret_number:
        print(" Congratulations! You guessed the correct number.")
        break
    else:
        print("Try again!")
```

# LOOPS: WHILE

It is possible to create a loop that never stops, in fact, its pretty easy to do, so be careful.

```
while True:  
    print("oopsie")
```

Crtl + c to stop it!



**CAREFUL: INFINITE WHILE LOOPS!!**

While loops can happen until the condition is no longer **true** OR there is a **break**.

Here, we are asking a question until

```
secret_number = 42
while True:
    user_guess = int(input("Guess a number between 1 and 100: "))
    if user_guess == secret_number:
        print(" Congratulations! You guessed the correct number.")
        break
    else:
        print("Try again!")
```

# LOOPS: WHILE

For / in loop can be used to iterate through each item in a list or in a range.

```
for item in a_list:  
    #do task
```

In this example, item is the name we are calling the individual item. a\_list is a reference to an actual list.

```
fruits = ['apple', 'banana', 'cherry']  
for fruit in fruits:  
    print(len(fruit))
```

# LOOPS: FOR IN

Fruit represents a single item in the list. It changes as we iterate through the list. Every loop, we're looking at the next item of fruit inside fruits.

```
fruits = ['apple', 'banana',  
         'cherry']
```

Fruits references  
the specific list

Fruit can be named anything, but this is typical naming convention.

```
for fruit in fruits:  
    print(len(fruit))
```

# BACK TO FOR / IN LOOPS

Range returns a value that can be used to move through a set number of iterations

Here range is 0 - 6 (non inclusive)

```
for x in range(6):  
    print(x)  
# 0 1 2 3 4 5
```

Here range is 3 - 6 (non inclusive)

```
for x in range(3, 6):  
    print(x)  
# 3 4 5
```

# LOOPS: FOR IN RANGE



With three values we can set the **increment per loop**, here I'm setting it to 2. So it will count by 2 (non inclusively)

```
for x in range(0, 6, 2):  
    print(x, end=" ")  
# 0 2 4
```

Note: you can use `end=", "` to indicate how the line ends instead of a new line. Ex; here we put a `", "` instead of a line break. This is useful when debugging loops!

```
for i in range(-1, 5, 2):  
    print(i, end=", ") # prints: -1, 1, 3,
```

# LOOPS: FOR IN RANGE

**Enumerate** is used to access both the **index** and **value** of each element. This is useful if we need to know the exact position within a list, not only the value. T

```
animals = ["lions", "tigers", "bears"]  
for i, animal in enumerate(animals):  
    print(f"{i} index contains {animal}")
```

You can use enumerate with an index in the for loop, or without. Without you will need to access each element individually.

```
for animal in enumerate(animals):  
    print(animal)  
    print(animal[0])  
    print(animal[1])
```



(0, 'lions') 0 lions  
(1, 'tigers') 1 tigers  
(2, 'bears') 2 bears

The diagram shows three arrows pointing from the code above to the output below. An orange arrow points from `print(animal)` to the first tuple `(0, 'lions')`. A red arrow points from `print(animal[0])` to the index `0`. A green arrow points from `print(animal[1])` to the value `'lions'`.

# LOOPS: FOR ENUMERATE

**For / in** gives us the **value** of each element in the list

```
for animal in animals:  
    print(animal, end=" ", "  
> lions, tigers, bears,
```

**Range** gives us the **number** within a range (here, the range is the length of the animals list, but this can be any range.

```
for animal in range(len(animals)):  
    print(animal, end=" ", "  
# 0, 1, 2,
```

**Enumerate** gives us the **index** and **value** for each element in the list

```
for i, animal in enumerate(animals):  
    print(f"{i} index contains {animal}")
```

```
0 index contains lions  
1 index contains tigers  
2 index contains bears
```

# LOOPS: FOR, RANGE, ENUMERATE

Data types

User input

Order of operations

Comparisons

Pseudo code

Doc strings

**OTHER THINGS:**

Coming up: Test

**NEXT CLASS:**