Lecture 7 Execution Control 1

- One of the fundamental aspects of a program is *execution control*.
- This lecture will introduce `if`, `if-else`, and `if-elif-else`, conditional statements which control execution in a program.
- This lecture will introduce the concept of repeated execution of a block of code in a **loop** with the `while` conditonal statement

In [1]:

```
import numpy as np
from numpy import random
from matplotlib import pyplot as plt
#
rng = random.default_rng(seed = 1121)
```

Conditional Statements: `if`

- Conditionals are statements that evaluate a **logical statement**, using the `if` statement, and then only execute a set of code if the condition evaluates as True.

In [2]:

```
condition = True
if condition:
    print('This code executes if the condition evaluates as True.')
```

```
This code executes if the condition evaluates as True.
```

In [3]:

```
# equivalent to above
if condition == True:
    print('This code executes if the condition evaluates as True.')
```

```
This code executes if the condition evaluates as True.
```

- If the conditional is not met, the code inside the `if` statement does not execute

In [4]:

```
sign = "unknown"
n = 1
if n > 0:
    sign = 'positive'
print(sign)
```

```
positive
```

In [5]:

```
sign = "unknown"
n = -1
if n > 0:
    sign = 'positive'
print(sign)
```

```
unknown
```

- Notice this version doesnt print anything! This is because the print command is inside the if statement (look at the indentation). And the condition is not met.

In [6]:

```
sign = "unknown"
n = -1
if n > 0:
    sign = 'positive'
    print(sign)
```

Syntax notes `if`

- There are important pieces to the synatax

1. the logical statement leads with an `if` statement
2. the logical statement is followed by a colon `:`
3. the code to be executed if the conditional statement is met is indented.

- Indentation facilitates clearly understanding of the hierarchy of execution control.

Table of Comparison Operations

- A logical statement almost always involves comparisons. Examples of comparisons that come to mind might be

1. `==` - equal
2. `!=` - not equal
3. `>` - greater than
4. `>=` - greater than or equal
5. `<` - less than
6. `<=` - less than or equal

Logical Operators for Execution Control

- The results of logical operations **MUST** return a *single* value of **True** or **False** can also be used in execution control with an if statement.
- Logical operations on arrays will often return Boolean arrays that contain the result of a comparison operator applied to each element of the array.
- These can be combined in execution control by

1. `np.any` - check if any of the of elements of an array are True
2. `np.all` - check if all of the elements of an array are True

In [7]:

```
# Example 1
array_a = rng.integers(-10,10,20)
print(array_a)
if np.any(array_a < 0):
    print('There were negative numbers')
```

```
[ -6  -8  -4   3   8   9  -1  -2   6  -1  -3   6   4  -6   3  -2   6  -9
 -10   5]
There were negative numbers
```

In [8]:

```
#what happens when the logical statement is false.
array_a = rng.integers(-10,10,20)
print(array_a)
#I changed any to all
if np.all(array_a < 0):
    print('All numbers were negative numbers')
#Nothing!
```

```
[  2   9   2  -3  -8   3  -3  -9   8  -7  -5  -3  -9   9   9  -6  -4  -8
 -10  -7]
```

Conditional statements: `else`

- After an if, you can use an `else` that will run if the logical statement was **False** and the conditional statement was not met.
- **Only one of the blocks of code will run**. The logical statement can only return one of **True** or **False**

In [9]:

```
condition = False
if condition:
    print('This code executes if the condition evaluates as True.')
else:
    print('This code executes if the condition evaluates as False')
```

```
This code executes if the condition evaluates as False
```

In [10]:

```
array_a = rng.integers(-10,10,20)
if np.all(array_a < 0):
    print('All numbers were negative numbers')
else:
    print('At least one number was a positive number')
```

```
At least one number was a positive number
```

In [11]:

```
array_a = rng.integers(-10,10,20)
if np.all(array_a < 0):
    print('All numbers were negative numbers')
else:
    print('At least one number was zero or a positive number')
    n_notnegative = np.sum(array_a >= 0)
    print('Not negative: ', n_notnegative)
```

```
At least one number was zero or a positive number
Not negative:  10
```

Syntax notes `else`

- Notice that the `else` statement is itself a conditional statement. Thus, it is completed with a colon `:`
- **implicit** - the complement of the if statement.
- Notice also, that the structure of an if-else statement can have multiple lines of code under each conditional statement
  if logical statement: do this #CODE BLOCK EXECUTES ONLY IF LOGICAL STATEMENT IS TRUE and this and this else: do this #CODE BLOCK EXECUTES ONLY IF LOGICAL STATEMENT IS FALSE and this and this

Conditional Statements: `elif`

- Multiple **non-overlapping** conditional statements can be organized together using an `elif` statement.
- `elif` combines else and if into one statement.

In [12]:

```
condition_1 = True
condition_2 = True

if condition_1:
    print('This code executes if condition_1 evaluates as True.')
elif condition_2:
    print('This code executes if condition_1 did not evaluate as True, but condition_2 does.')
else:
    print('This code executes if both condition_1 and condition_2 evaluate as False')
```

```
This code executes if condition_1 evaluates as True.
```

- Notice that the block of code above never evaluated condition_2, because the evaluations are in serial order.
- once the conditional in the if statement is **True** *the remaining statements are never evaluated*

In [13]:

```
condition_1 = False
condition_2 = True

if condition_1:
    print('This code executes if condition_1 evaluates as True.')
elif condition_2:
    print('This code executes if condition_1 did not evaluate as True, but condition_2 does.')
else:
    print('This code executes if both condition_1 and condition_2 evaluate as False')
```

```
This code executes if condition_1 did not evaluate as True, but condition_2 does.
```

- In this case, because condition_1 is **False** condition_2 is tested, and evaluates **True**.

In [14]:

```
condition_1 = False
condition_2 = False

if condition_1:
    print('This code executes if condition_1 evaluates as True.')
elif condition_2:
    print('This code executes if condition_1 did not evaluate as True, but condition_2 does.')
else:
    print('This code executes if both condition_1 and condition_2 evaluate as False')
```

```
This code executes if both condition_1 and condition_2 evaluate as False
```

- Now we make it to the `else`.

`elif` without an `else`

- An `else` statement is not required, but if both the `if` and the `elif` condtions are not met (both evaluate as **False**), then nothing is returned.

In [15]:

```
condition_1 = False
condition_2 = True

if condition_1:
    print('This code executes if condition_1 evaluates as True.')
elif condition_2:
    print('This code executes if condition_1 did not evaluate as True, but condition_2 does.')
```

```
This code executes if condition_1 did not evaluate as True, but condition_2 does.
```

In [16]:

```
condition_1 = False
condition_2 = False

if condition_1:
    print('This code executes if condition_1 evaluates as True.')
elif condition_2:
    print('This code executes if condition_1 did not evaluate as True, but condition_2 does.')
```

- `elif` after an `else` does not make logical sense since `else` is the complement of `if`
- The order will always be `if-elif-else` ...with only the `if` being required.
- If the `elif` is at the end...it will never be tested, as the else will have already returned a value once reached (and thus Python will throw an error).

In [17]:

```
## THIS CODE WILL PRODUCE AN ERROR
condition_1 = False
condition_2 = False

if condition_1:
    print('This code executes if condition_1 evaluates as True.')
else:
    print('This code executes if both condition_1 and condition_2 evaluate as False')
elif condition_2:
    print('This code executes if condition_1 did not evaluate as True, but condition_2 does.')
```

```
  Cell In[17], line 9
    elif condition_2:
    ^
SyntaxError: invalid syntax
```

- Don't trust python to find your mistakes.
- Python will not always produce an error and will frequently allow you to write nonsense.

In [18]:

```
if 1+1 == 2:
    print("I did Math")
elif 1/0:
    print("I broke Math")
else:
    print("I didn't do math")
# Python is an interpreted language.  it is not testing all your code before executing.
# It is interpeting it line by line while executing it.
```

```
I did Math
```

In [19]:

```
if 1/0:
    print("I did Math")
elif 1+1 == 2:
    print("I broke Math")
else:
    print("I didn't do math")
# Python is an interpreted language.  it is not testing all your code before executing.
# It is interpeting it line by line while executing it.
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[19], line 1
----> 1 if 1/0:
      2     print("I did Math")
      3 elif 1+1 == 2:

ZeroDivisionError: division by zero
```

Syntax notes `elif`

```
if logical statement 1:
    do this  #CODE BLOCK EXECUTES ONLY IF LOGICAL STATEMENT 1 IS TRUE
    and this
    and this
elif logical statement 2: #THIS ONLY EVALUATES IF LOGICAL STATEMENT 1 IS FALSE
    do this  #CODE BLOCK EXECUTES ONLY IF LOGICAL STATMENT 2 is TRUE
    and this
    and this
else:
    do this  #CODE BLOCK EXECUTES ONLY IF LOGICAL STATEMENT 1 and LOGICAL STATEMENT 2 IS FALSE
    and this
    and this
```

- An important implication of using the `if-elif-else` for execution control is to have a clear understanding of the relationship between logical statement 1 and logical statement 2.
- You should think about this in terms of sets and subsets. There is a subset that meets the conditions of logical statement 1. If that is **True** , *logical statement 2 is never evaluated*
- Thus *implicitly* in the above bit of code, the `elif` statement should be read (in your mind) as
  elif (logical statement 2) & ~ (logical statement 1)

This is a logically screwed up set of statements. DONT DO THIS!

In [20]:

```
n = 6
if n < 10:
    print('single digit')
elif n > 5:
    print('greater than 5')
else:
    print('double digit number')
```

```
single digit
```

- The problem with the above block of code is that a subsets of integers meet the criterion overlap.
- if n is 6,7,8,9 it evaluates **TRUE** for `if` and **TRUE** for `elif`, but only the `if` block executes.
- Also, if n >= 10 it meets the `elif` criteria the complement of the `if` and only the first one executes.

Properties of conditionals

- All conditionals start with an `if`, can have an optional and variable number of `elif`'s and an optional `else` statement
- Conditionals can take any expression that can be evaluated as `True` or `False`.
- At most one component (`if` / `elif` / `else`) of a conditional will run
- The order of conditional blocks is always `if` then `elif`(s) then `else`
- Code is only ever executed if the condition is met
- The first condition met is executed. Other condiitons will not be evaluated.

Compound conditional statements

- The only requirement on the logical statement that makes the conditional `if` statement is that can return either **True** or **False**.
- We can make compound conditional statements using Boolean Operators `&` (and) `|` (or)

1. & - (and) to require two (or more) Conditional Statements are True
2. | - (or) to require that at least one of two (or more) Contional Statements are True

- When you do this you **must** place each individual comparison operator **inside parenthesis**.

In [21]:

```
n = 8
if (n%2 == 1) | (n > 10):
    print('Either odd or Larger than 10')
else:
    print('Even and less than or equal to 10')
```

```
Even and less than or equal to 10
```

- Note that the opposite of an OR (|) conditional boolean operator is an AND (&).

In [22]:

```
n = 17
if (n%2 == 0) & (n > 10):
    print('Even and Larger than 10')
else:
    print('Either odd or less than or equal to 10')
```

```
Either odd or less than or equal to 10
```

- Note that the opposite of an AND (&) conditional boolean operator in an OR (|).

Nested conditional statements

- Another option for execution control with combined conditions is to nest `if-elif-else` statements.
- This structure allows for more flexible options in the output than the compound conditional statement.

In [23]:

```
n = 17
if n%2 == 0:   #Everything inside here is even
    if n > 10:
        print('Even and larger than 10')
    else:
        print('Even and smaller than or equal to 10')
else:          #Everything inside here is not even, i.e., odd.
    if n > 10:
        print('Odd and larger than 10')
    else:
        print('Odd and smaller than or equal to 10')
```

```
Odd and larger than 10
```

Syntax notes - nested conditional statements

- Notice that if you nest conditional statements you have to be careful about indentation. Indentation determines which conditional statement is associated with the code block.

```
  if logical statement 1:
      if logical statement 2: # CODE BLOCK EXECUTES ONLY IF LOGICAL STATEMENT 1 AND LOGICAL
STATEMENT 2 IS TRUE
          do this
          and this
          and this
  else: # CODE BLOCK EXECUTES ONLY IF LOGICAL STATEMENT 1 IS TRUE AND LOGICAL STATEMENT 2 IS
FALSE
          do this
          and this
          and this
  else:
      if logical statement 2: # CODE BLOCK EXECUTES ONLY IF LOGICAL STATEMENT 1 IS FALSE AND
LOGICAL STATEMENT 2 IS TRUE
          do this
          and this
          and this
      else: # CODE BLOCK EXECUTES ONLY IF LOGICAL STATEMENT 1 IS FALSE AND LOGICAL STATEMENT 2 IS
FALSE
          do this
          and this
          and this
```

When do I need to use if-elif-else?

- It is rare to need to use execution control in data analysis or visualization.
- Usually if you do that you've written bad code that runs slowly and failed to use array methods.
- I would be disappointed.
- But is very common in programs that control experiments.

- Let's make a grading example. Suppose I want to write a bit of code to assign a letter grade.
- grades between

1. 87.5 and 100 are A
2. 75 and 87.5 are B
3. below 75 are C

In [24]:

```
grade = 85
if grade > 87.5:
    lettergrade = 'A'
elif grade > 75:
    lettergrade = 'B'
else:
    lettergrade = 'C'
print(grade,' ',lettergrade)
```

```
  85   B
```

In setting up if-elif-else statements its quite important to think through the order of the conditional statements.

Execution Control with **Loops**

- Loops are a fundamental construct of execution control. Loops are a way of controlling *repetition* of code, i.e., allow the same block of code to run multiple times.

`While` Loops

- A `while` loop is a type of loop that runs as long as a *logical statement* is **True**.
- When the logical condition becomes **False**, the code stops running.
- The general form of a while loop in Python is below:

1. The while loop begins with a logical statement that tests a **variable** which initially returs be of value **True**
2. There is an indented block of code inside the while loop.
3. Within that code block there must be something that updates the value of the **variable**

```
while logical_statement:
    do this
    and that
    and that
    AND YOU MUST DO SOMETHING THAT POSSIBLY CHANGES
    THE STATE OF THE LOGICAL STATEMENT TO FALSE
```

- Eventually, after one or more repetitions of the block of code, the logical statement returns a value **False** and the block of code no longer executes.

In [25]:

```
# Example 1
i = 0   #since i is the variable tested n the logical statement, it must be set to an initial value
while i<4:  # this is the conditional statement which controls execution
    print(i)
    i = i+1 #this is a critical line, as it updates the value of i
```

```
0
1
2
3
```

CRITICAL STEP IN WHILE LOOPS

- There are three critical pieces to constructing a while loop.

1. The variable(s) that will be used in the logical statement must be initialized to some value.
2. A `while` statement performs execution control based on a logical statement that tests the variables being **True**.
3. Inside the `while` loop, the variable(s) used in the logical statement must eventually be updated to a value that evaluates **False**

- When making use of a `while` loop it is critical that there always be a line inside the while loop that updates whatever is being tested by the logical statement.
- If that line is missing, the while loop will become an **infinite** loop and the code will only stop by an act of ***VIOLENCE*** by you against the code block, your VS Code instance, or in a worst case, against your computer (hard reboot).

Boolean Indicator Control

- Sometimes, its easy to think about this by using a Boolean Indicator variable to control the while loop.
- Here I rewrite the loop above with an indicator variable.

In [26]:

```
i = 0
keep_looping = True
while keep_looping:
    print(i)
    i = i+1
    if i >= 4:
        keep_looping = False
```

```
0
1
2
3
```

Counters

- In many instances, we want to keep track of the number of times the block of code is executed by the loop using a counter.

Example Sum of Random Numbers to a Limit.

- This example shows a classical type of problem we often encounted in modeling and simulation.
- We want to execute some code until a variable reaches a critical value.

In [27]:

```
limit = 100  #set the limit
total = 0  #start at 0
nsamples = 0 #start a counter to keep track of the number of samples
while total < limit:
    sample = rng.integers(0,10) #get one random number from a normal distribution
    total = total + sample  # this updates the value of total
    nsamples = nsamples + 1 # this keeps track of the number of samples taken
print('total = ',total)
print('nsamples = ',nsamples)
```

```
total =  101
nsamples =  26
```

- I could save a bit more information from this loop, by actually saving out the running values of total on each step.

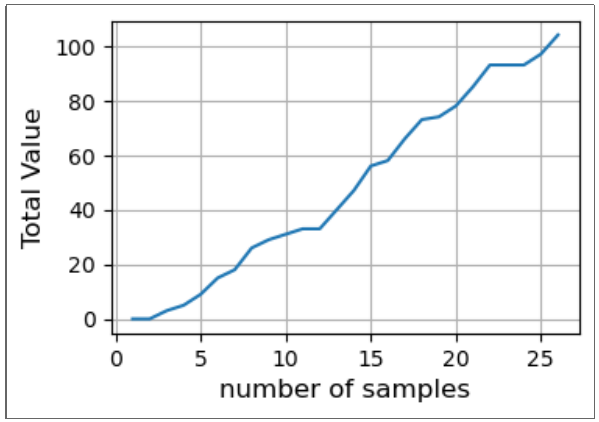
In [28]:

```
limit = 100  #set the limit
total = 0  #start at 0
totallist = list() # a list to keep track of the values on each iteration of the loop.
nsamples = 0 #start a counter to keep track of the number of samples
while total < limit:
    nsamples = nsamples + 1 # this keeps track of the number of samples taken
    sample = rng.integers(0,10) #get one random number from a normal distribution
    total = total + sample  # this updates the value of total
    totallist.append(total) # I append the current value of total to the list.
print('total = ',total)
print('nsamples = ',nsamples)
print('totallist = ', totallist)
```

```
total =  104
nsamples =  26
totallist =  [0, 0, 3, 5, 9, 15, 18, 26, 29, 31, 33, 33, 40, 47, 56, 58, 66, 73,
74, 78, 85, 93, 93, 93, 97, 104]
```

In [29]:

```
numberofsamples = np.arange(1,nsamples+1)
fig = plt.figure(figsize = (3,2)) # I selected the figure dimension here
ax = fig.add_axes([0,0,1,1])
ax.plot(numberofsamples,totallist)  # i use the quick an dirty way to make a plot here
ax.set_xlabel('number of samples',fontsize = 12)
ax.set_ylabel('Total Value',fontsize = 12)
plt.grid(True)
plt.show()
```



Example When will I have a million dollars?

- Suppose you have 20000 (or 2E04) dollars. You put in the bank, and each year they give you 5% interest (compounded annually).
- How many years will it take for you to have more than one million (1000000 or 1E06) dollars.

In [30]:

```
### What is the answer to the question? In terms of the code?
interest = 0.05 # interest rate
balance = 20000 # starting balance 1E04
nyears = 0
target = 1000000 # 1E06
while balance <= target: #Execute the block as long as my balance is less than or equal to target!
    balance = balance+interest*balance
    nyears = nyears+1
print('Balance of ', balance, 'after ', nyears, 'years')
```

```
Balance of  1040790.2624036912 after  81 years
```

- This type of **accumulator** model will show up in Cognitive Science classes on Decision Making.