Lecture Execution Control with `For` Loops

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

In [3]:

```
import numpy as np
from numpy import random
#Lets start the random number generator
rng = random.default_rng(seed = 9876)
from matplotlib import pyplot as plt
```

Decoding the problem

The hardest part of using various forms of execution control is decoding the problem. That is, you have to look at the problem, as words in human terms, and map it onto an **algorithm** that you can implement on a computer.

- The bank balance problem

Compute the annual balance of a bank account with an initial deposit of $10000 and annually compounded interest at a fixed rate of 5% until the balance reaches a target. We wannt to know the annual balance, and the number of years to reach the target.

- The bank balance problem

Compute the annual **balance** of a bank account with an initial **deposit** of $10000 and annually compounded interest at a fixed **rate** of 5% until the balance reaches a **target**. We want to know the annual balance, and the number of **years** to reach the **target**.

In [4]:

```
interest = 0.05 # interest rate
balance = 10000 # starting balance 1E04
annual = list()
annual.append(balance)
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
    annual.append(balance)
```

In [5]:

```
print(nyears) #number of years
print(balance) # final balance
```

```
95
1030346.7644609454
```

`for` and `while` loop

- Two repetion structures in Python are `for` loops and `while` loops.
- `for` loops run a set number of times. In a typical application, a block of code needs to be applied to the elements of a list or an array or list one at a time. *The number of repetitions of the code is determined in advance*.
- `while` loops run as long as a specific logical condition is true. The key difference between `while` and `for` loops is that *the number of repetitions in a while loop is unknown*

In [6]:

```
#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
```

- To express the same operations as a `for` loop I need to think about the possible set of values I want **i** to take.
- Then, I want to ask python to repeat the `print` statement for each value of **i**

In [7]:

```
values = [0,1,2,3]  #these are the numbers I want to print. I made a list here.
for i in values:
    print(i)
```

```
0
1
2
3
```

- The output is identical.
- The critical difference is that I know exactly what values i want the variable i to take in a `for` loop.
- In a 'while' loop, I use a **logical statement** on the variable i to determine how many times to run it.

`For` Loop

- A `for` loop is a repetition structure where a code block runs a specified number of times.
  for var in the_range_of_var: do this and this and this
- **the_range_of_var** can be given in many forms. The most common way is the `range` command, but it can also be a **list** or a numpy **array** as we will discuss further below.
- The most important difference between a `for` loop and `while` loop is that a `for` loop *automatically increments the value of* **var** *to the next element in the_range_of_var* in each repetition of the code block
- As we will see in the examples below **var** is often used an index variable that indexes into arrays sequentially.

The `range` Function

- creates an integer list spanning a *range*. Very useful for constructing `for` loops.

In [8]:

```
for i in range(4):
    print(i)
```

```
0
1
2
3
```

- Python's `range` function can be customized by supplying up to three arguments. The general format of the range function is below:
  range(start,stop,step)
- *start*, *stop*, *step* must be **integers**

In [9]:

```
#here range(start,stop,step) - start is 0 (inclusive) stop is 6 (exclsuive) and 2 is the step size.
for i in range(0,6,2):
    print(i)
```

```
0
2
4
```

- `range` can even count down if necessary

In [10]:

```
for i in range(5,-6,-1):
    print(i)
```

```
5
4
3
2
1
0
-1
-2
-3
-4
-5
```

- To examine the contents of `range`, convert to a list or a numpy array before printing.
- Once you get comfortable using `range`, you wont need to do this.

In [11]:

```
x = list(range(0,6,2))
print(x)
y = np.array(range(0,6,2))
print(y)
```

```
[0, 2, 4]
[0 2 4]
```

Loops over lists and arrays

- `for` loops can take a list or array for the variable to iterate over in the loop.

In [12]:

```
mylist = ['I ','wish ','Kawhi ','had ','better ','knees']
# the other way to do it is to use range over the length of the list (4):
for i in mylist:
    print(i)
```

```
I
wish
Kawhi
had
better
knees
```

- Iterating over a list or array usually is done by using an index variable.
- Even in python, its important to understand how this works because you often need to iterate over multiple arrays at the same time.
- To do this, instead of the `for` loop iterating over a variable, it will iterate over the *index* into the array.

In [13]:

```
mylist = ['I ','wish ','Kawhi ','had ','better ','knees']
nlist = len(mylist) #I know its 4, but its good to automatically get this.
for i in range(nlist): # I use range(nlist) to make i go over the values 0,1,2,3 which are index into mylist.
    print(mylist[i]) #here I index into mylist and then print.
```

```
I
wish
Kawhi
had
better
knees
```

- Why is this way of thinking about it useful. Here is a simple example.

In [14]:

```
names = ['a','b','c','d']
grades = [95,82,90,88]
nstudents = len(names)
# the len function gets you the length of the list names
for istudent in range(nstudents):
    #istudent is my indexing variable
    print(names[istudent],' : ',grades[istudent])
    # i use istudent as an index into both names and grades.
```

```
a  :  95
b  :  82
c  :  90
d  :  88
```

- `enumerate` is a useful python function for making `for` loops over a list

In [15]:

```
for index, item in enumerate(names):
    print(item)
    print(index)
```

```
a
0
b
1
c
2
d
3
```

In [16]:

```
#when using enumarate, you keep track of two things, the index into the list, and the item in the list:
for index, item in enumerate(names):
    print(item, ' : ', grades[index])
```

```
a  :  95
b  :  82
c  :  90
d  :  88
```

Indexing Loops

- The most useful way to think about `for` loops is like this,
  for index in Variable_to_iterate_over
- `index` is the key to the for loop. Inside the `for` loop `index` allows you to

1.iterate over the elements of a list or array that must be operated on.

```
y = x[index]**2
```

1. systematically increase the value of a variable,

```
y = 2*index+1
```

2. iterate over multiple lists or arrays in parallel.

```
z = y[index] + x[index]
```

3. control indexing into an output array.

```
y[index] = x[index]**2
```

Organizing input and output arrays with a for loop.

- These are not the best use cases, but I chose these, because they should feel familiar.

- Example: Projecting your bank account after 10 years
- Suppose you have $500 in your bank account and interest rate is 6.0% compounded annually.
  Write a for loop that will compute how much you will have in **nyears**.

In [17]:

```
nyears = 10
# number of years
interest = 0.06 # interest rate
balance = 500 # starting balance
for year in range(1,nyears+1): #notice I do nyears+1 here.  This is because year 0 is the first value and then
                               # I want to interate nyears more years.
    balance = balance+interest*balance
print('Balance of ', balance, 'after ', nyears, 'years')
```

```
Balance of  895.4238482714267 after  10 years
```

- If I want to keep track of the balance after each year, I could use

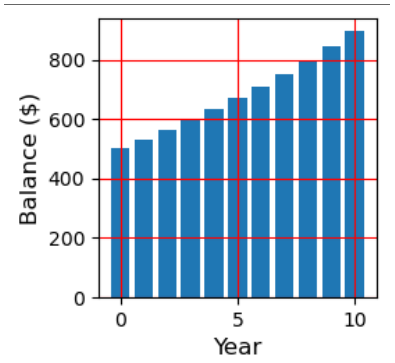
In [18]:

```
nyears = 10# number of years
balance = list()
interest = 0.06 # interest rate
balance.append(500) # starting balance at year 0
years_list = [0]
for year in range(1,nyears+1):  #again, notice I use range with nyears and my index variables is years,
    newbalance = balance[year-1]+interest*balance[year-1]
    # notice I used years -1.
    #to get year n, I have to calculate using the balance from n-1
    balance.append(newbalance)
    years_list.append(year)
print('balance = ', balance)
print('years = ', years_list)
```

```
balance =  [500, 530.0, 561.8, 595.5079999999999, 631.2384799999999, 669.1127887999999, 709.
2595561279999, 751.8151294956799, 796.9240372654207, 844.739479501346, 895.4238482714267]
years =  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [19]:

```
### Let's make a nice plot of it.
fig = plt.figure(figsize = (2,2))
a = fig.add_axes([0,0,1,1])
a.bar(years_list,balance) # notice I control the x axis variable here using np.arange to go from 0 to 5
a.set_xlabel('Year',fontsize = 12)
a.set_ylabel('Balance ($)',fontsize=12)
plt.grid('on',color='r')
plt.show()
```



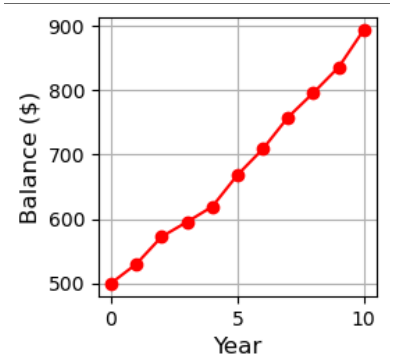
- What if interest rate varies over the next 10 years? So that the interest rate (percentage) for each of the five years is is [6,8,4,4,8,6,7,5,5,7]

In [20]:

```
interest = np.array([0.06,0.08,0.04,0.04,0.08,0.06,0.07,0.05,0.05,0.07]) # interest rate
nyears = np.size(interest) # I determined the number of years from the lenght of interest using the size functio
balance = list()
balance.append(500) # starting balance
years_list = [0]
for year in range(1,nyears+1):
    newbalance =  balance[year-1]+interest[year-1]*balance[year-1]
    #notice that I use year here as an index into interest
    #and into balance
    #again I use year -1 to get the previous years data.
    balance.append(newbalance)
    years_list.append(year)
```

In [21]:

```
### Let's make a graph again
fig = plt.figure(figsize = (2,2))
a = fig.add_axes([0,0,1,1])
a.plot(years_list,balance,'ro-') # notice I control the x axis variable here using np.arange to go from 0 to 11
a.set_xlabel('Year',fontsize = 12)
a.set_ylabel('Balance ($)',fontsize=12)
plt.grid('on')
plt.show()
```



- Lets consider doing this problem with arrays instead of lists.
- Lets go back to simple case of a single interest rate first.
- when working with arrays, we have to preallocate the array

In [22]:

```
nyears = 10# number of years
#when working with arrays, we have to preallocate the arra
balance = np.zeros(nyears+1)
#I need 1 more year because I start at year 0
interest = 0.06 # interest rate
balance[0] = 500 # starting balance at year 0
years_list = np.arange(nyears+1)
for year in range(1,nyears+1):  #again, notice I use range with nyears and my index variables is years,
    balance[year] = balance[year-1]+interest*balance[year-1]
    # notice I am using indexing in the output and input side of the calculation.
print('balance = ', balance)
print('years = ', years_list)
```

```
balance =  [500.         530.         561.8        595.508      631.23848
 669.1127888  709.25955613 751.8151295  796.92403727 844.7394795
 895.42384827]
years =  [ 0  1  2  3  4  5  6  7  8  9 10]
```