CLASS 14: CONDITIONAL STATEMENTS & LOOPS IN PYTHON

ENGR 102 – Introduction to Engineering

Conditional Statements

- `if` statements
- Logical and relational operators
- `if…else` statements

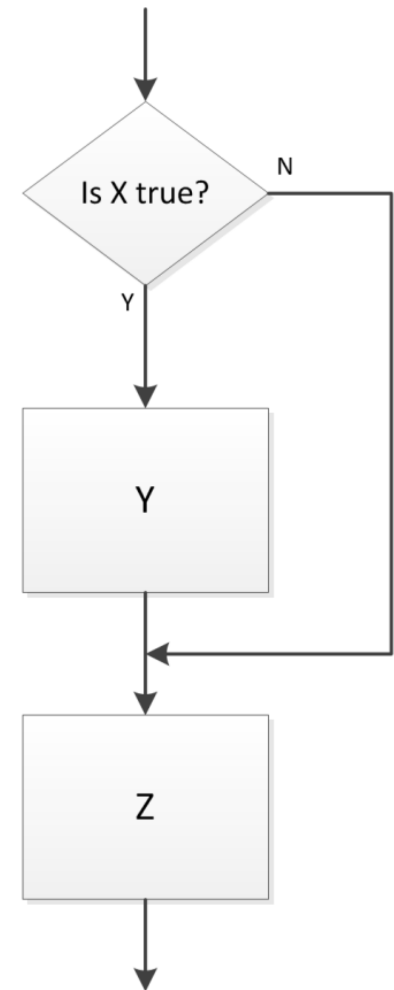
The `if` Statement

- We've already seen the ***if structure***
  - If X is true, do Y, if not, don't do Y
  - In either case, then proceed to do Z
- In Python:

```
if condition:
    statements
        ⋮
```

- *Statements* are executed ***if*** *condition* is ***True***
  - Statement block defined by ***indenting*** those lines of code
- *Condition* is a ***logical expression***
  - Boolean - either True or False
  - Makes use of ***logical and relational operators***
- May use a ***single line*** for a single statement:

```
if condition: statement
```



Logical and Relational Operators

Operator	Relationship or Logical Operation	Example
==	Equal to	x == b
!=	Not equal to	k != 0
<	Less than	t < 12
>	Greater than	a > -5
<=	Less than or equal to	7 <= f
>=	Greater than or equal to	(4+r/6) >= 2
and	AND – ***both*** expressions must evaluate to true for result to be true	(t > 0) and (c == 5)
or	OR – ***either*** expression must evaluate to true for result to be true	(p > 1) or (m > 3)
not	NOT– negates the logical value of an expression	not (b < 4*g)

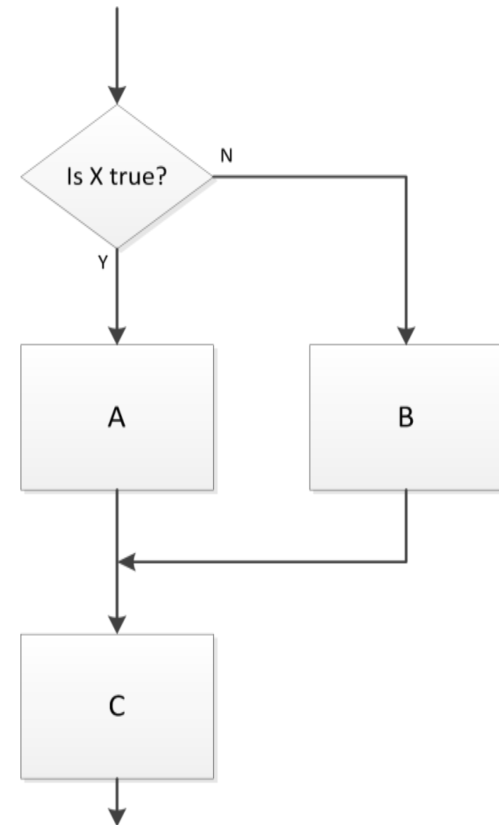
The `if`…`else` Structure

- The ***if … else structure***
  - Perform one process if a condition is true
  - Perform another if it is false

- In Python:

```
if condition:
  statements_1
else:
  statements_2
```

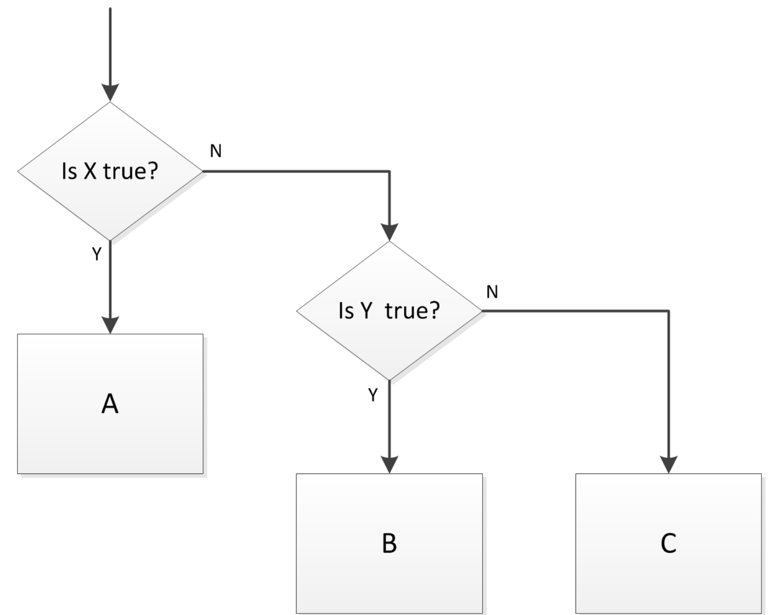
- Note that `if` and `else` code blocks are defined by ***indents***



The `if`…`elif`…`else` Structure

- The ***if … elif … else structure***
  - If a condition evaluates as false, check another condition
  - May have an arbitrary number of ***elif*** statements
- In Python:

```
if condition₁:
  statements₁
elif condition₂:
  statements₂
else:
  statements₃
```



The `if...else`, `if...elif...else` Structures

- Some examples:

```
 9
10  if (t >= 0) and (p > 8):
11      x = p**2 * t
12      y = 3*q + p
13  else:
14      x = 0
15      y = q + p**2
16
```

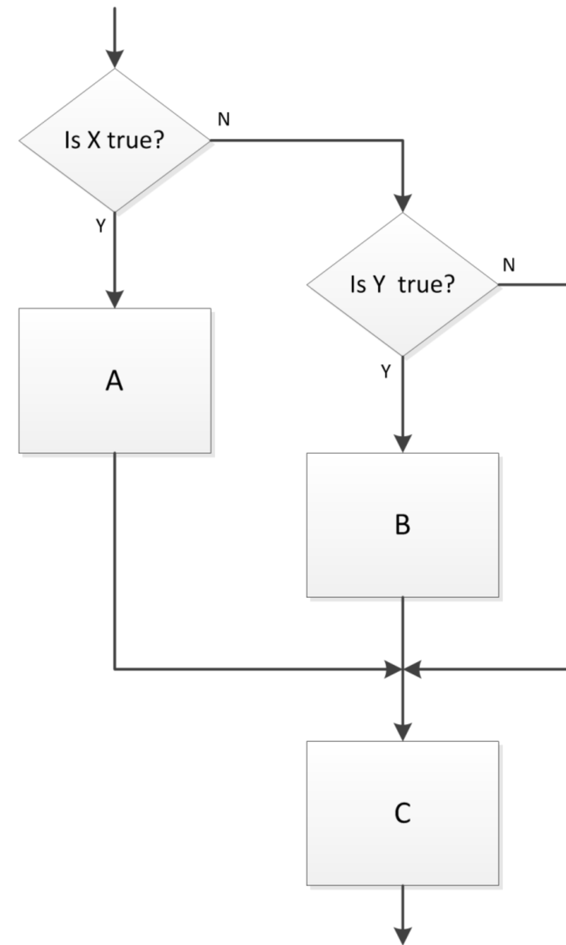
```
17
18  if x == 0:
19      f = 2*np.pi
20  elif x <= -1:
21      f = np.pi/4
22  elif (y != 436) or (x > 18):
23      f = 0
24  else:
25      f = 2*np.pi/3
26
```

- Note that code blocks are defined by indents
  - Each line must have the same indent - use the Tab key
  - Meaningful whitespace is a distinguishing characteristic of Python
  - Other languages use brackets or *end* statements

The `if…elif` Structure

- We can have an **`if`** statement without an `else`
- Similarly, an **`if…elif`** structure need not have an **`else`**
- In Python:

```
if condition₁:
  statements₁
elif condition₂:
  statements₂
```



if…elif…else

Exercise

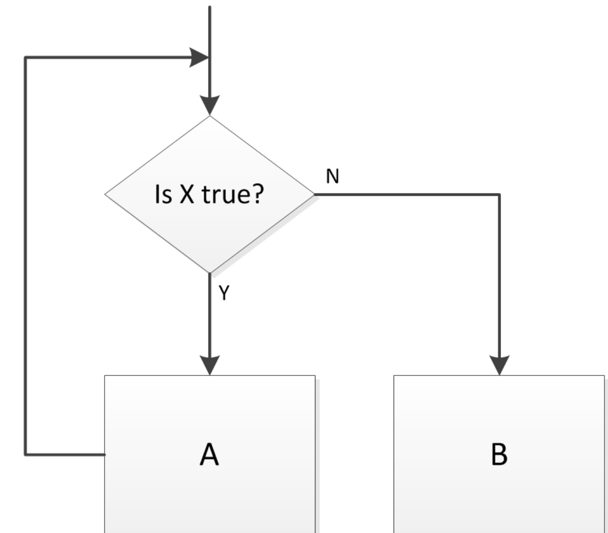
- Write a Python script to implement the following pseudocode:
  - `Prompt user to input an integer, store as x`
  - `If x < 5, print "The number is less than 5."`
  - `Else, if x < 10, print "The number is between 5 and 10."`
  - `Otherwise, print "The number is at least 10."`

10

`while` Loops

The `while` loop

- The ***while loop***
  - *While* X is true, do A
  - Once X becomes false, proceed to B



- In Python:

```
while condition:
    statements
        ⋮
```

- *Statements* are executed as long as *condition* remains true
  - *Condition* is a ***logical expression***
- ***Whitespace*** (indent) defines while block

while Loop – Example 1

- Consider the following while loop example
  - Repeatedly increment x by 7 as long as x is less than or equal to 30
  - Value of x is displayed on each iteration

```
7   # increment a number by 7 until it exceeds 30
8
9   x = 12
10
11  while x <= 30:
12      x = x + 7
13      print(x)
14
```

- x values displayed: `19, 26, 33`
- x gets incremented beyond 30
  - All loop code is executed as long as the condition was true at the ***start of the loop***

The break Statement

- Let's say we don't want x to increment beyond 30
  - Add a conditional break statement to the loop

```
18  # increment a number by 7 and exit the loop before it exceeds 30
19  x = 12
20
21  while x <= 30:
22      if (x+7)>30:
23          break
24      x = x + 7
25      print(x)
```

- `break` statement causes loop exit before executing all code
- Now, if `(x+7)>30`, the program will break out of the loop and continue with the next line of code
- `x` values displayed: 19, 26
- For nested loops, a break statement breaks out of the current loop level only

`while` Loop – Example 1

- The previous example could be simplified by modifying the while condition, and not using a `break` at all

```
30  # or, change the while condition so that x will not increment beyond 30
31
32  x = 12;
33
34  while (x+7) <= 30:
35      x = x + 7
36      print(x)
37
```

- Now the result is the same as with the `break` statement
  - x values displayed: 19, 26
- This is not always the case
  - The `break` statement can be very useful
  - May want to break based on a condition other than the loop condition
- `break` works with both `while` and `for` loops

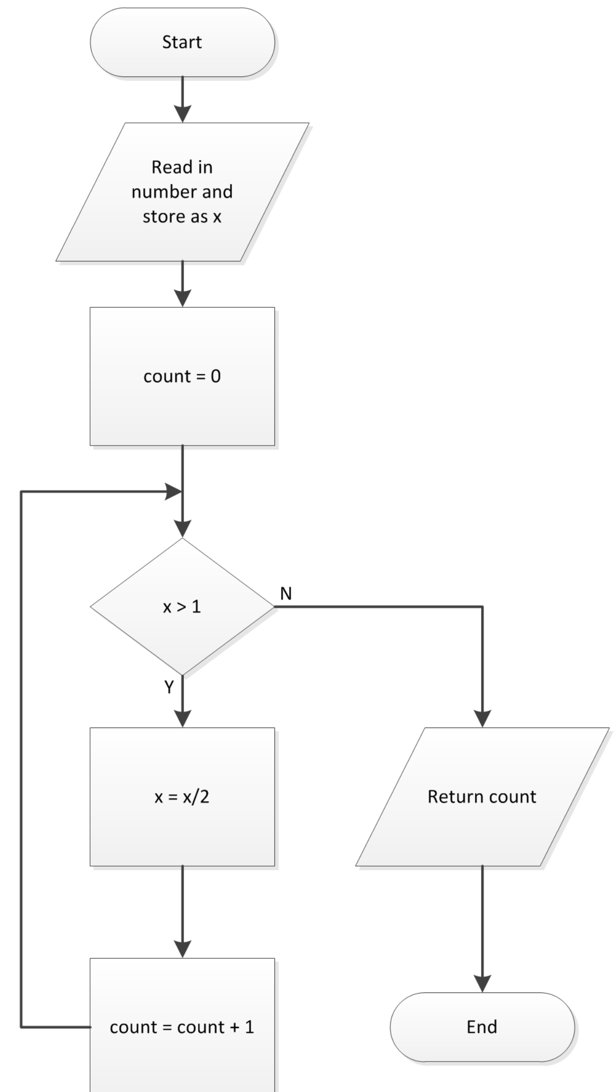
while Loop – Example 2

- Next, let's revisit the while loop examples from Section 4
- Use `input()` to prompt for input
- Use `print()` to return the result

```
39  # determine how many times a number
40  # must be halved to get a result <= 1
41
42  x = input('Enter a number: ');
43  x = float(x)
44
45  count = 0;
46
47  while x > 1:
48      x = x/2
49      count = count + 1
50
51  print('count = {:d}'.format(count))
```

```
Enter a number: 130
count = 8

In [42]:
```



while Loop – Example 3

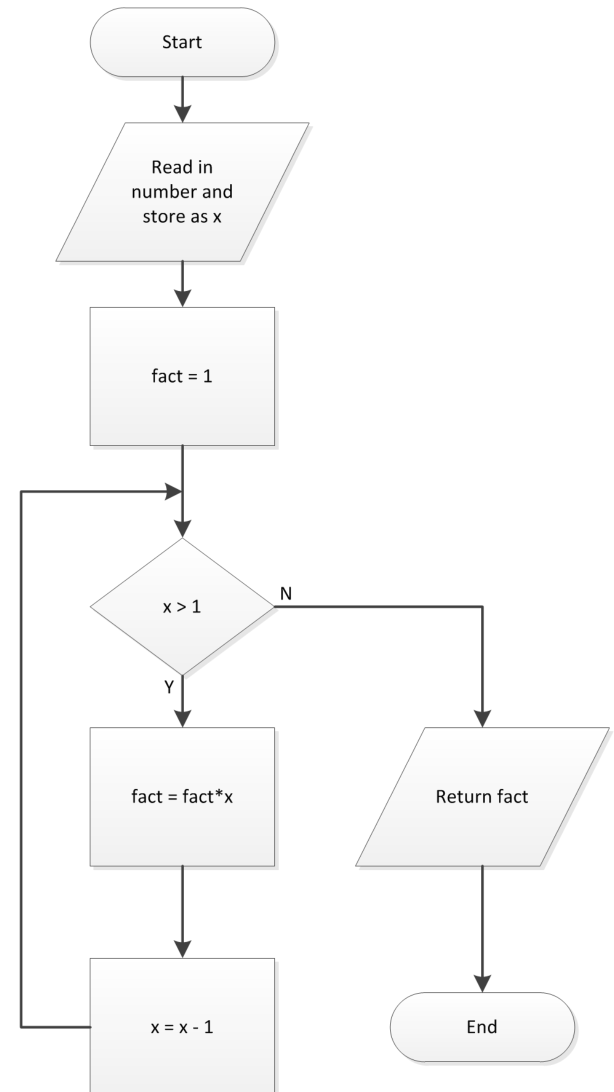
- Here, we use a `while` loop to calculate the factorial value of a specified number

```
54  # calculate factorial(x)
55
56  x = input('Enter an integer: ')
57  x = xin = float(x)
58
59  fact = 1
60
61  while x > 1:
62      fact = fact*x
63      x = x - 1
64
65
66  print('\nfact({}) = {}'.format(xin, fact))
```

```
Enter an integer: 12

fact(12.0) = 479001600.0

In [52]:
```



`while` Loop – Example 3

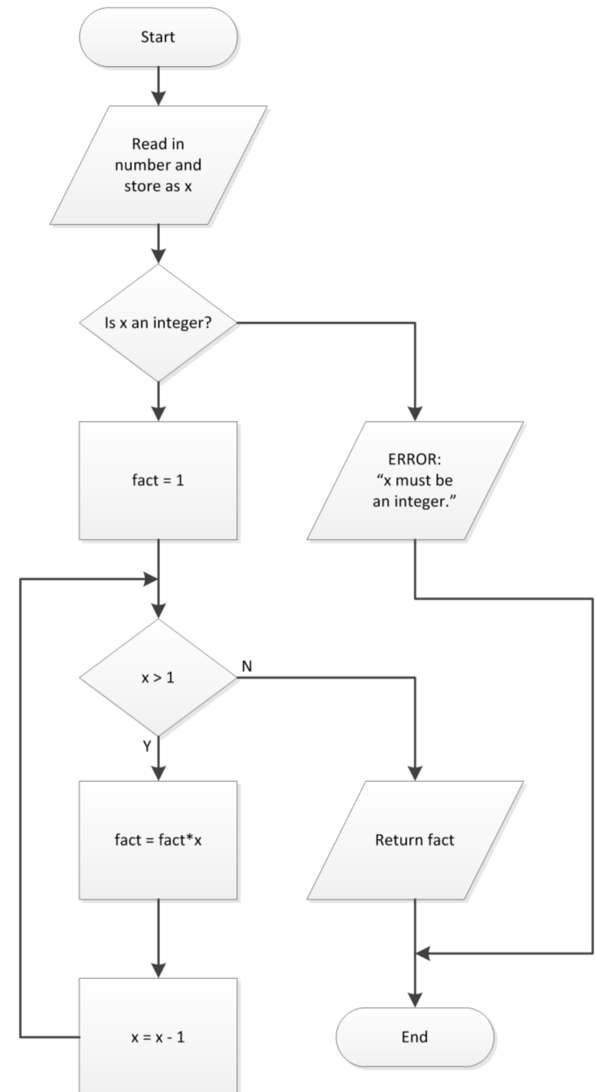
- Add error checking to ensure that x is an integer
- One way to check if x is an integer:

```
69  # calculate factorial(x)
70  # with error checking for integer input
71
72  x = input('Enter an integer: ')
73  x = float(x)
74
75  # check if x is an integer
76  if x != int(x):
77      raise Exception('ERROR: x must be an integer.')
78
79  fact = 1
80
81  while x > 1:
82      fact = fact*x
83      x = x - 1
84
85  print('\nfact({:d}) = {:d}'.format(xin, fact))
```

```
Enter an integer: 11.5
Traceback (most recent call last):

  File "C:\Users\webbky\Box\KWebb\Classes\ENGR102_103\Notes\Python\
    raise Exception('ERROR: x must be an integer.')

Exception: ERROR: x must be an integer.
```



while Loop – Example 3

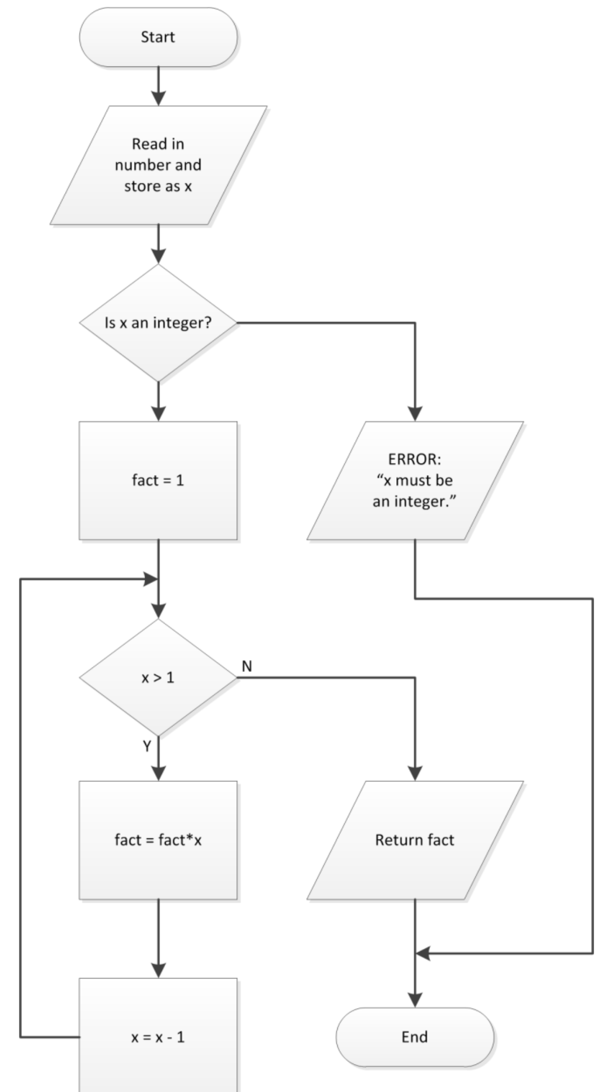
- Another possible method for checking if x is an integer:

```
88   # calculate factorial(x)
89   # alternative way to check for an integer input
90
91   x = input('Enter an integer: ')
92   x = float(x)
93
94   # check if x is an integer
95   if (x - np.floor(x)) != 0:
96       raise Exception('ERROR: x must be an integer.')
97
98   fact = 1
99
100  while x > 1:
101      fact = fact*x
102      x = x - 1
103
104  print('\nfact({:d}) = {:d}'.format(xin, fact))
```

```
Enter an integer: 20.3
Traceback (most recent call last):

  File "C:\Users\webbky\Box\KWebb\Classes\ENGR102_103\Notes\Python\
    raise Exception('ERROR: x must be an integer.')

Exception: ERROR: x must be an integer.
```



Infinite Loops

- A loop that never terminates is an ***infinite loop***
- Often, this unintentional
  - Coding error
- Other times infinite loops are intentional
  - E.g., microcontroller in a control system
- A while loop will never terminate if the while condition is always true
  - By definition, `True` is always true:

```
while True:
  statements repeat infinitely
```

while True

- The `while True` syntax can be used in conjunction with a `break` statement, e.g.:

- Useful for multiple break conditions
- Control over break point
- Could also modify the while condition

```
43    while True:
44        iter = iter + 1        # increment iteration index
45
46        xrold = xr             # store previous estimate for error approx
47
48        # Choose upper or lower sub-interval as next bracketing interval
49        if (func(xl)*func(xr)) >= 0:        # root is in upper sub-interval
50            xl = xr
51
52        if (func(xu)*func(xr)) >= 0:        # root is in lower sub-interval
53            xu = xr
54
55        if xl == xu:         # func(xr) == 0, exactly (unlikely)
56            epsa = 0
57        else:
58            # update the root estimate
59            xr = xu - func(xu)*(xu - xl)/(func(xu) - func(xl))
60            # approximate the error
61            epsa = abs((xr-xrold)/xr)*100
62
63        # check if stopping criterion is satisfied or if maximum number of
64        # iterations has been reached
65        if (epsa<=reltol):
66            break
67        elif (iter >= maxiter):
68            print('\nMaximum # of iterations reached - exiting.\n\n')
69            break
70
71    fxr = func(xr);
72
73    return [xr, fxr, epsa, iter]
```

while Loop

Exercise

- Write a Python script to implement the following pseudocode:
  - Import Python's time module
  - Prompt user to input a timer value in seconds, store as t.
  - Use a while loop to count down from t to zero:
    - Display t
    - Decrement t by one
    - Wait for 1 sec
  - Display "Time's up!"

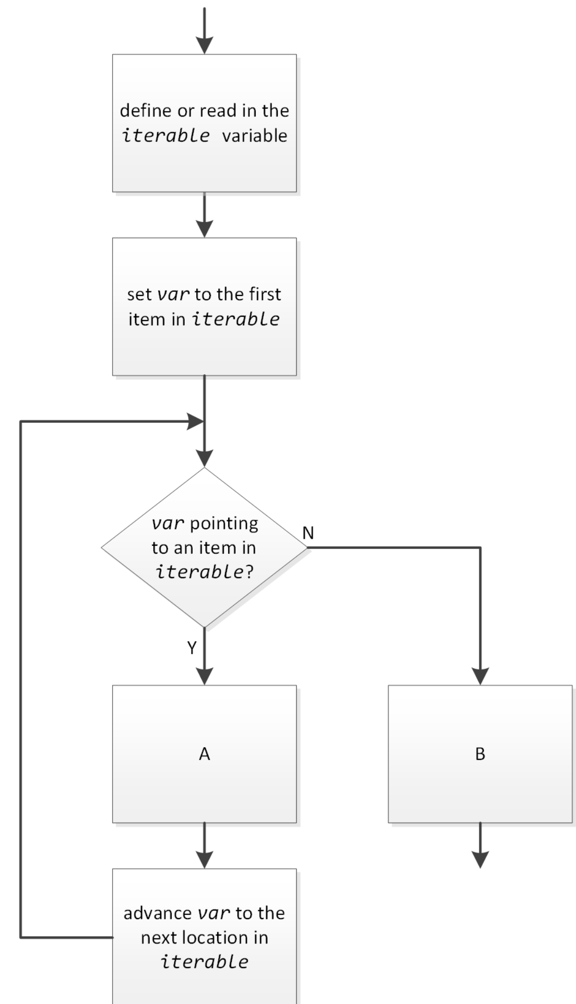
for Loops

The for Loop

- The ***for loop***
  - Loop executed a specified number of times

```
for var in iterable:
    statements
        ⋮
```

  - *iterable*: any ***iterable*** object (ndarray, list, tuple, dict, str)
  - *var*: variable that assumes each successive value in *iterable* on each iteration
  - *Statements*: code block that is executed once for each item in *iterable*
- Inherently ***collection-based***, not counter-based
  - Iterates through each item in a collection
  - But, can be counter-based, as we will see



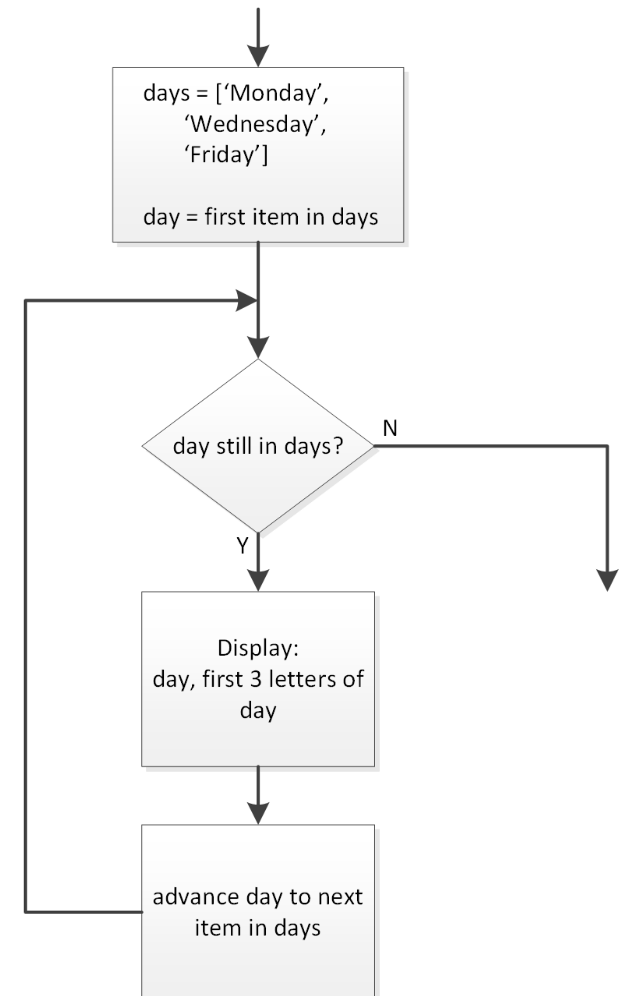
for Loop – Example 1

- A ***collection-based*** (or ***iterator-based***) for loop
  - Iterates through each value in a list of days
  - No explicit loop counter

```
 7    days = ['Monday',
 8            'Wednesday',
 9            'Friday']
10
11    print('\n')
12
13    for day in days:
14        print(day, ', ', day[0:3])
15
```

```
Monday ,  Mon
Wednesday ,  Wed
Friday ,  Fri

In [70]:
```



for Loop – Example 2 – range( )

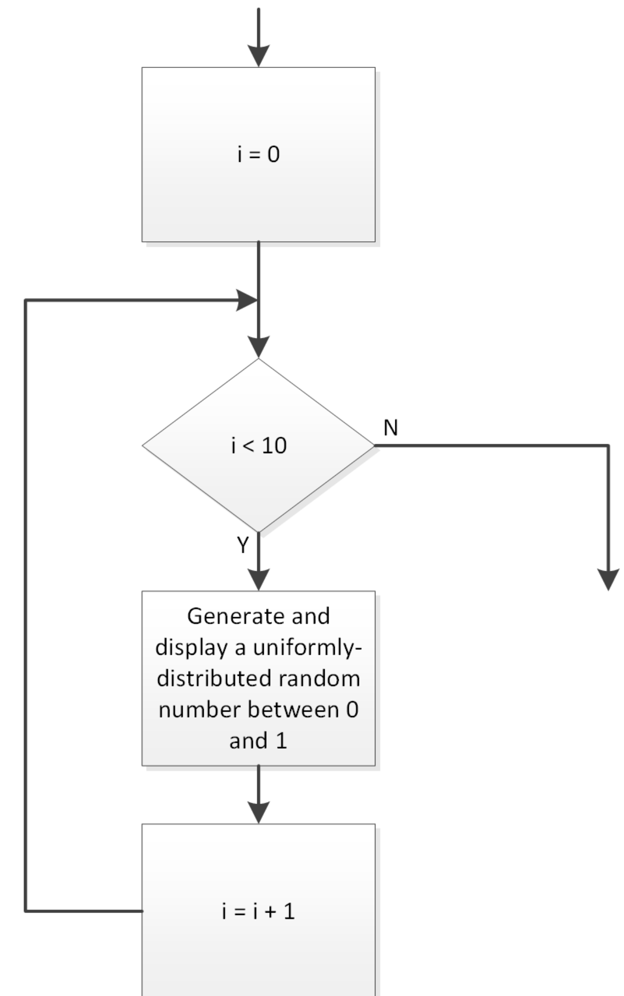
- ***Counter-based*** for loop
  - Use Python's `range( )` function:

    `range(start, stop, step)`

  - Generate a list of loop counter values to iterate through
  - Technically, still collection-based

```
19    rng = np.random.default_rng()
20
21    print('\n')
22
23    for i in range(10):
24        x = rng.uniform(low=0, high=1)
25        print('x = {:0.4f}'.format(x))
```

```
x = 0.0735
x = 0.2565
x = 0.0224
x = 0.5613
x = 0.1624
x = 0.2274
x = 0.9905
x = 0.6892
x = 0.7598
x = 0.7589
```



`for` Loop – Example 3 – `enumerate()`

- Sometimes we may want a combination of a collection-based and counter-based `for` loop
  - Iterate over both the ***values*** and ***indices*** of all items in an iterable
  - Use Python's **`enumerate()`** function
    - Generates an (index, value) pair for each item in the iterable
- For example, consider a list of numbers:

```
x = [2, 4, 6, 8]
```

- Generate (index, value) pairs for each item in x:

```
enumerate(x)
```

- Generates the following (index, value) pairs:

```
(0, 2), (1, 4), (2, 6), (3, 8)
```

- Can iterate over these (index, value) pairs with a `for` loop

for Loop – Example 3 – `enumerate()`

- Loop through an array of numbers to find the maximum value and its index
  - Use `enumerate()` to simultaneously loop through array values and their indices

```
29
30    x = rng.integers(0, 100, 10)
31    xmax = x[0]
32    imax = 0
33
34    for i, xval in enumerate(x[1:]):
35        if xval > xmax:
36            xmax = xval
37            imax = i+1
38
39    print('\nx = ', x)
40    print('\nxmax: x[{:d}] = {:d}'.format(imax, xmax))
41
```

```
x =  [63 65 57 17 69 63 92 47 86 73]

xmax: x[6] = 92

In [33]:
```