SECTION 5: STRUCTURED PROGRAMMING IN PYTHON

ENGR 103 – Introduction to Engineering Computing

Conditional Statements

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

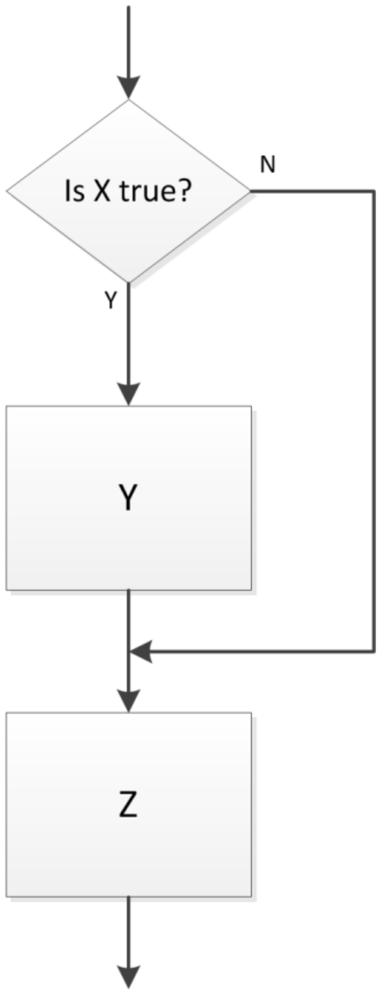
The `if` Statement

- 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

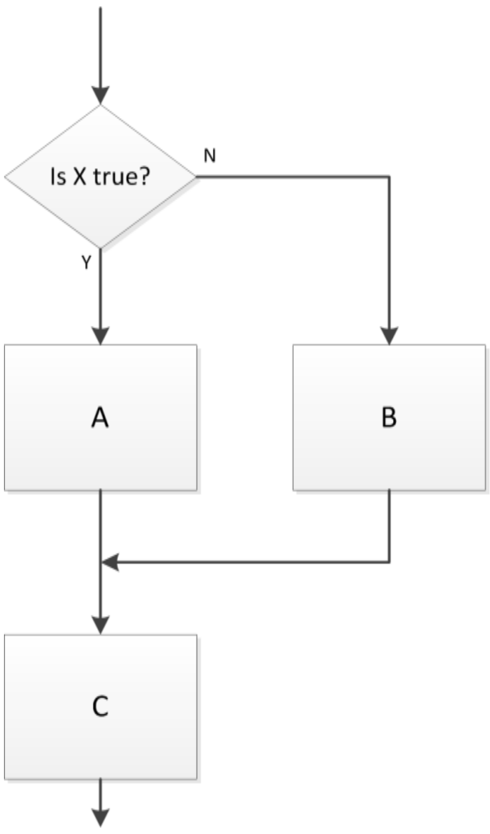
Operator	Relationship or Logical Operation	Example
`==`	Equal to	`x == b`
`!=`	Not equal to	`k != 0`
`<`	Less than	`t < 12`
`>`	Greater than	`a > -5`
`<=`	Less than or equal to	`7 <= f`
`>=`	Greater than or equal to	`(4+r/6) >= 2`
`and`	AND – ***both*** expressions must evaluate to true for result to 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:
  statements1
else:
  statements2
```

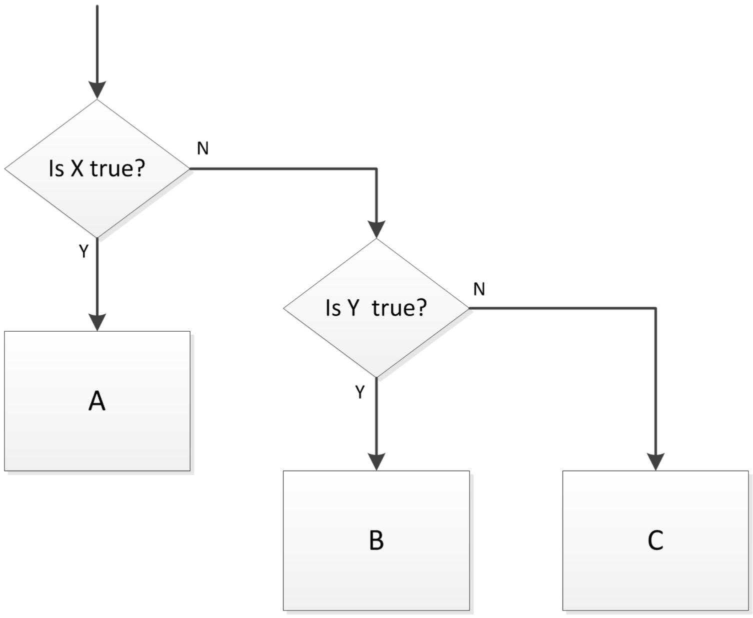
- Note that `if` and `else` code blocks are defined by ***indents***



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:

```
if (t >= 0) and (p > 8):
    x = p**2 * t
    y = 3*q + p
else:
    x = 0
    y = q + p**2
```

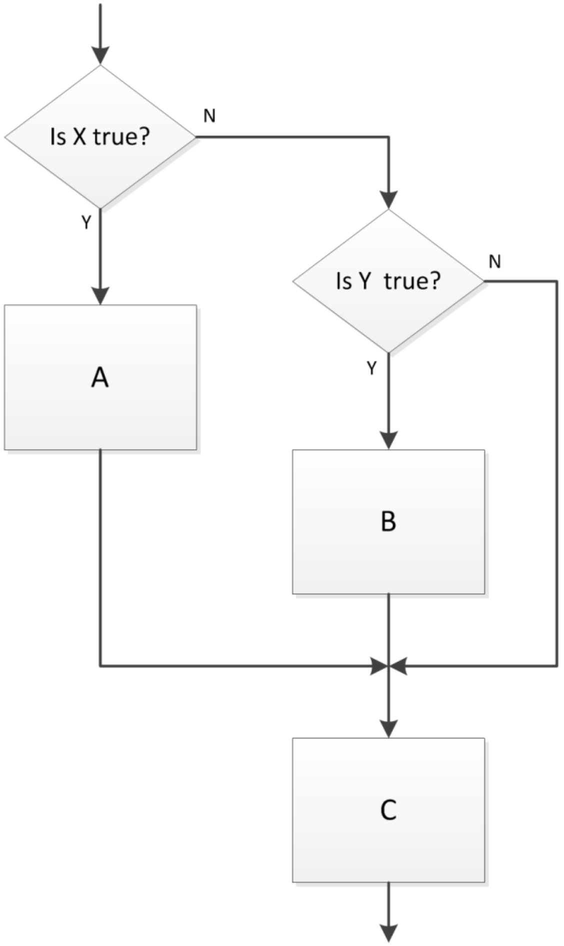
```
if x == 0:
    f = 2*np.pi
elif x <= -1:
    f = np.pi/4
elif (y != 436) or (x > 18):
    f = 0
else:
    f = 2*np.pi/3
```

- 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:

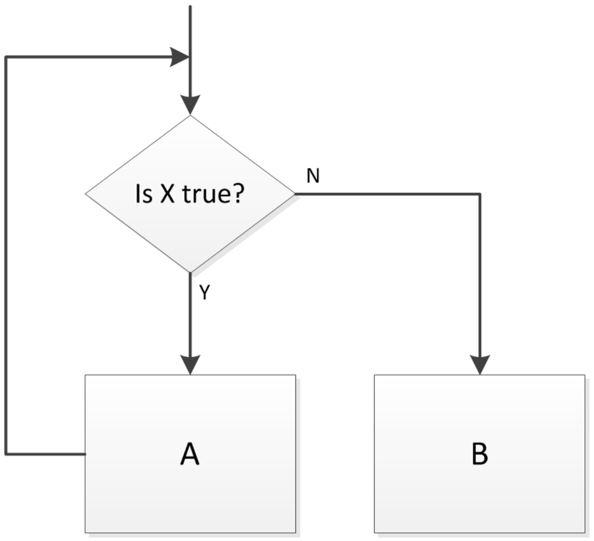
```
if condition1:
  statements1:
elif condition2:
  statements2
```



`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

```
# increment a number by 7 until it exceeds 30

x = 12

while x <= 30:
    x = x + 7
    print(x)
```

- 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

```
# increment a number by 7 and exit the loop before it exceeds 30
x = 12

while x <= 30:
    if (x+7)>30:
        break
    x = x + 7
    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

```python
# or, change the while condition so that x will not increment beyond 30

x = 12;

while (x+7) <= 30:
    x = x + 7
    print(x)
```

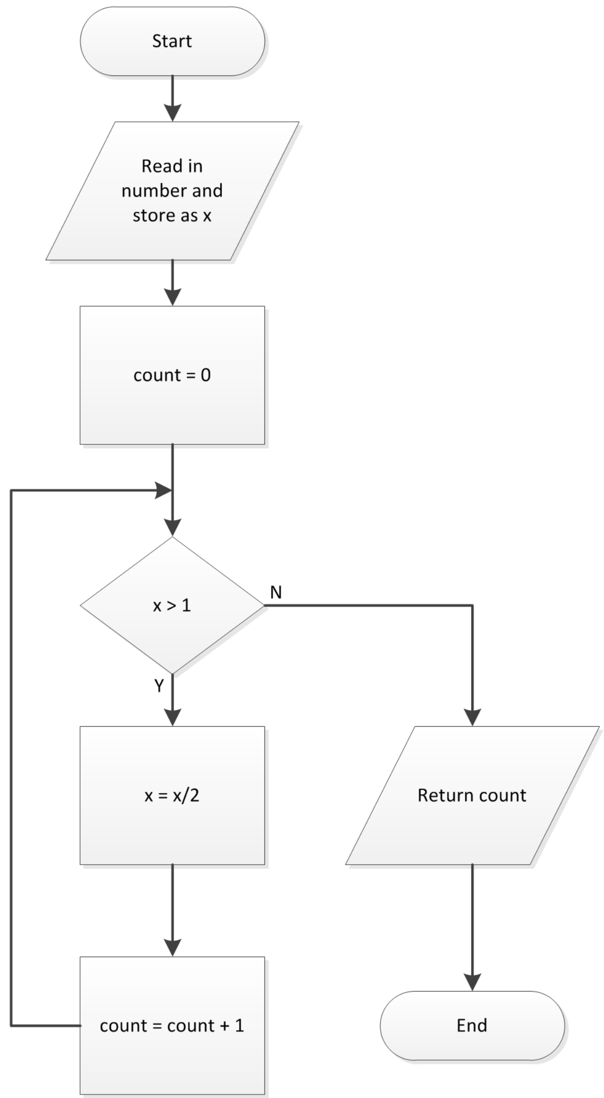
- Now the result is the same as with the `break` statement
  - x values displayed: 19, 26
- This is not always the case
  - The `break` statement can be very useful
  - May want to break based on a condition other than the loop condition
- `break` works with both `while` and `for` loops

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

```
# determine how many times a number
# must be halved to get a result <= 1

x = input('Enter a number: ');
x = float(x)

count = 0;

while x > 1:
    x = x/2
    count = count + 1

print('count = {:d}'.format(count))
```

```
Enter a number: 130
count = 8

In [42]:
```

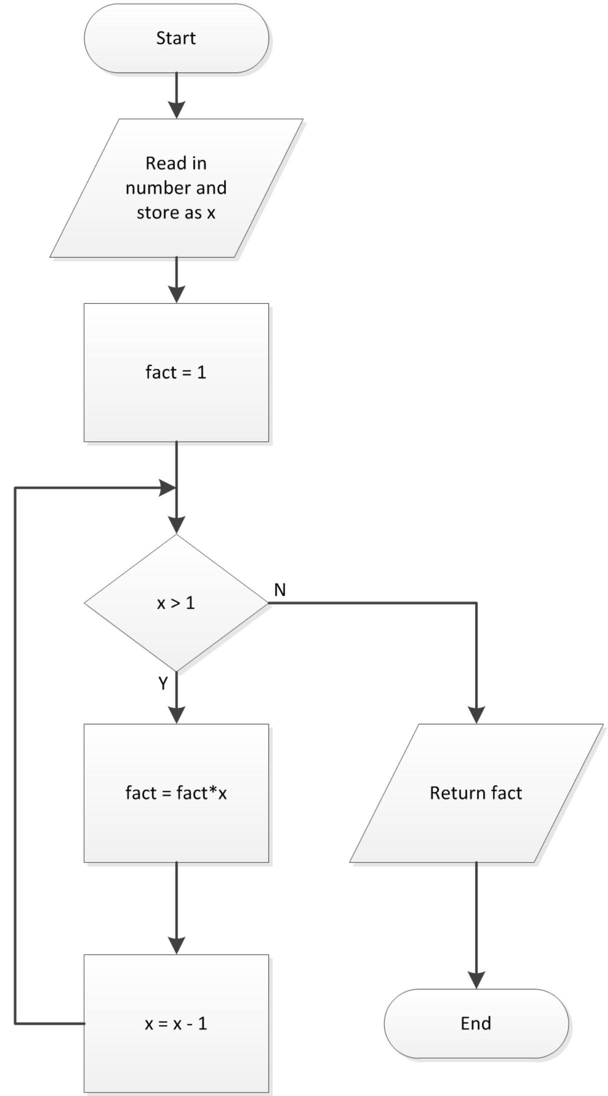


while Loop – Example 3

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

```python
# calculate factorial(x)

x = input('Enter an integer: ')
x = xin = float(x)

fact = 1

while x > 1:
    fact = fact*x
    x = x - 1


print('\nfact({}) = {}'.format(xin, fact))
```

```
Enter an integer: 12

fact(12.0) = 479001600.0

In [52]:
```

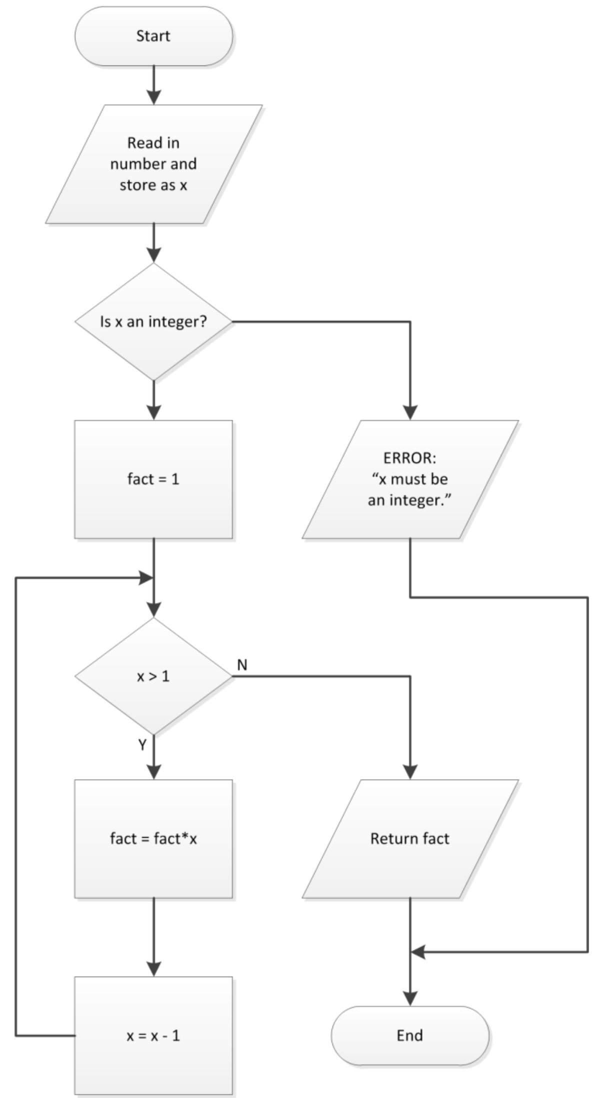


`while` Loop – Example 3

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

```python
# calculate factorial(x)
# with error checking for integer input

x = input('Enter an integer: ')
x = float(x)

# check if x is an integer
if x != int(x):
    raise Exception('ERROR: x must be an integer.')

fact = 1

while x > 1:
    fact = fact*x
    x = x - 1

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.
```

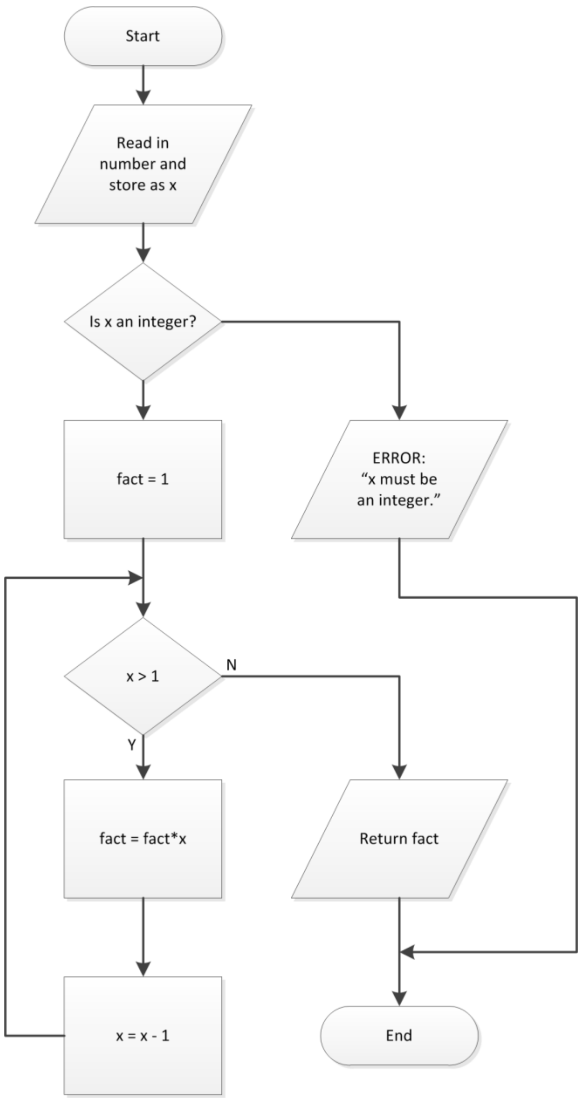


while Loop – Example 3

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

```python
# calculate factorial(x)
# alternative way to check for an integer input

x = input('Enter an integer: ')
x = float(x)

# check if x is an integer
if (x - np.floor(x)) != 0:
    raise Exception('ERROR: x must be an integer.')

fact = 1

while x > 1:
    fact = fact*x
    x = x - 1

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

```
    while True:
        iter = iter + 1        # increment iteration index

        xrold = xr             # store previous estimate for error approx

        # Choose upper or lower sub-interval as next bracketing interval
        if (func(xl)*func(xr)) >= 0:        # root is in upper sub-interval
            xl = xr

        if (func(xu)*func(xr)) >= 0:        # root is in lower sub-interval
            xu = xr

        if xl == xu:        # func(xr) == 0, exactly (unlikely)
            epsa = 0
        else:
            # update the root estimate
            xr = xu - func(xu)*(xu - xl)/(func(xu) - func(xl))
            # approximate the error
            epsa = abs((xr-xrold)/xr)*100

        # check if stopping criterion is satisfied or if maximum number of
        # iterations has been reached
        if (epsa<=reltol):
            break
        elif (iter >= maxiter):
            print('\nMaximum # of iterations reached - exiting.\n\n')
            break

    fxr = func(xr);

    return [xr, fxr, epsa, iter]
```

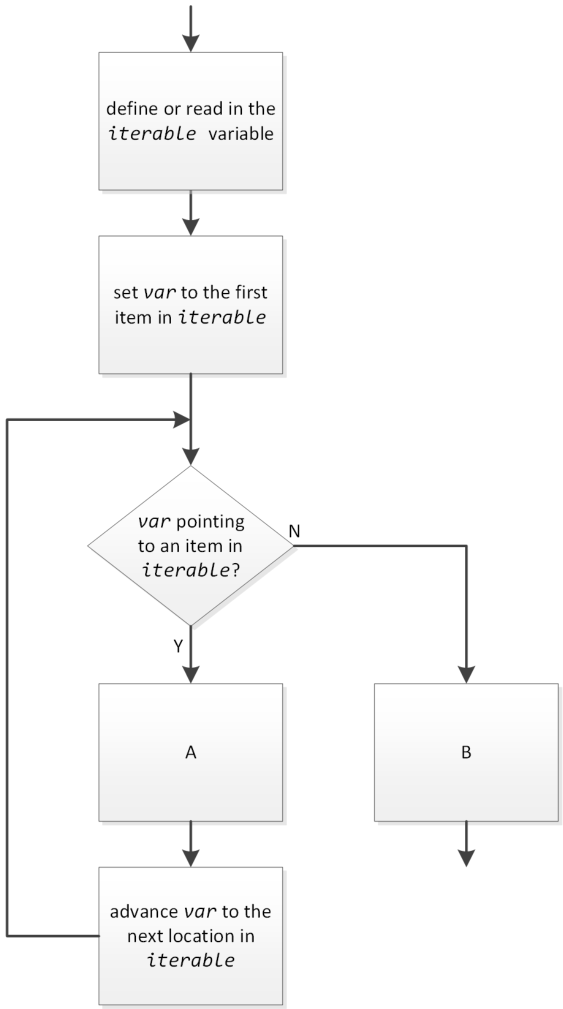
for Loops

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*
- ***Collection-based***, not counter-based
  - Iterates through each item in a collection
  - Can be counter-based, like flowchart to the right

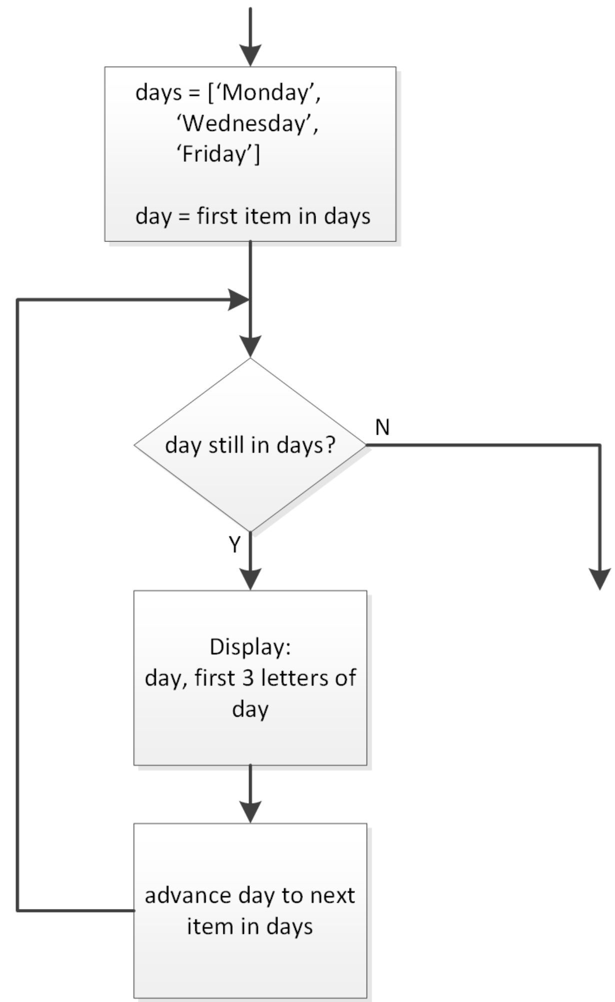


for Loop – Example 1

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

```
days = ['Monday',
        'Wednesday',
        'Friday']

print('\n')

for day in days:
    print(day, ', ', day[0:3])
```

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

In [70]:
```



for Loop – Example 2 – range( )

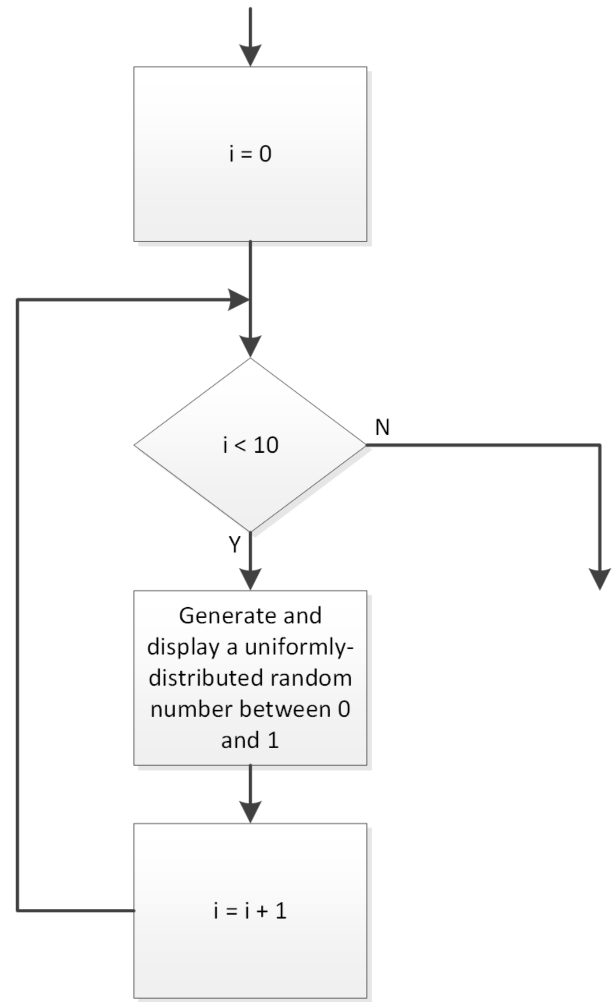
- ***Counter-based*** for loop
  - Use Python's `range( )` function:

    `range(start, stop, step)`

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

```python
rng = np.random.default_rng()

print('\n')

for i in range(10):
    x = rng.uniform(low=0, high=1)
    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, 10]
```

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

```
i, val = enumerate(x)
```

- Generates the following (i, val) pairs:

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

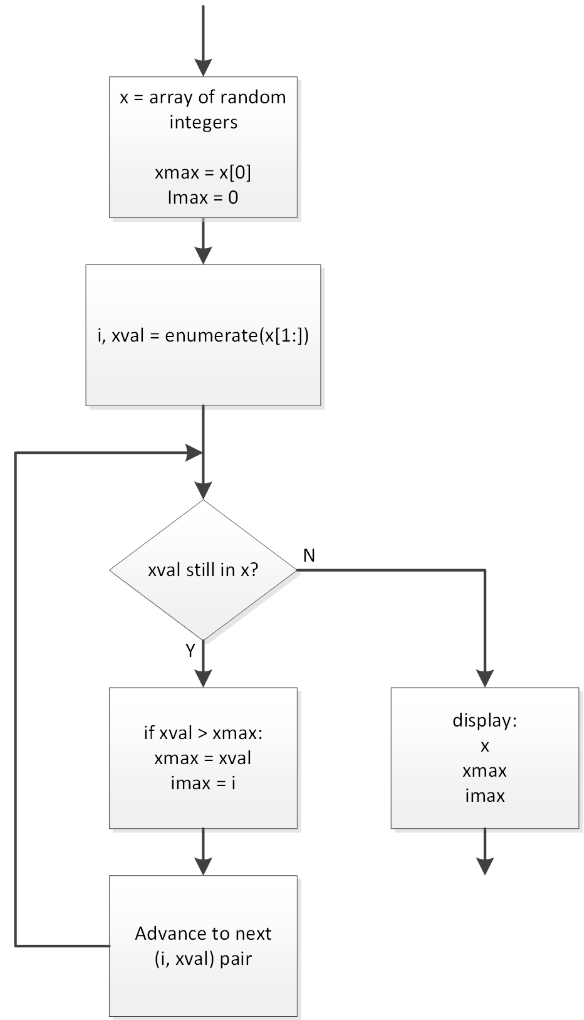
- Can iterate over these (index, value) pairs with a `for` loop

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

```python
x = rng.integers(0, 100, 10)
xmax = x[0]
imax = 0

for i, xval in enumerate(x[1:]):
    if xval > xmax:
        xmax = xval
        imax = 1

print('\nx = ', x)
print('\nxmax: x[{:d}] = {:d}'.format(i, xmax))
```

```
x =  [69 91 50 65 92 79 84 61 33 30]

xmax: x[8] = 92

In [131]:
```



Exercise – `for` Loop, `enumerate()`

Exercise

- The step response of a first-order system is given by

$$y(t) = 1 - e^{-\frac{t}{\tau}}$$

- Write a script to do the following:
  - Generate an array of τ values:

$$\tau = [1.0 \quad 1.5 \quad 2.0 \quad 2.5 \quad 3.0] \; sec$$

  - Generate a time vector with 2000 values between 0 and $5 * \max(\tau)$
  - In a ***for loop***, using the ***enumerate*** function, iterate through the values in τ and:
    - Calculate $y(t)$
    - Store the result as one column of a matrix, y
  - Outside of the for loop, plot each of the columns of y on a single set of axes

Nested Loops

Nested Loop – Example 1

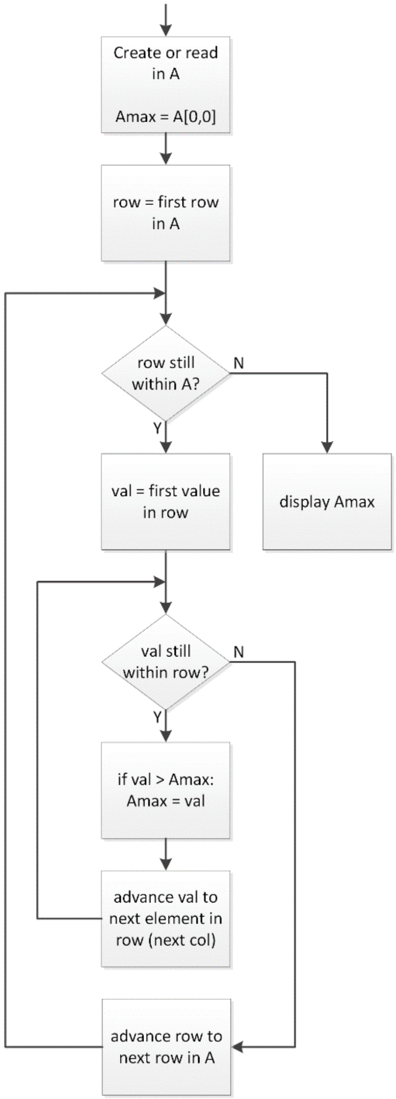
- Use a nested for loop to find the maximum value in a matrix or 2-D array
  - ***Outer loop*** steps through rows
  - ***Inner loop*** steps through columns
  - Store the largest value seen as the maximum value
- Consider an (m×n) matrix, A
  - **`A[0]`** ***indexes the first row***, so

```
for row in A:
```

  - Steps through the rows in A one-by-one
    - `row = A[0]`, `row = A[1]`, up to `row = A[-1]`
  - An inner loop steps through each element in each row

```
for row in A:
    for val in row:
        <code to check for max>
```

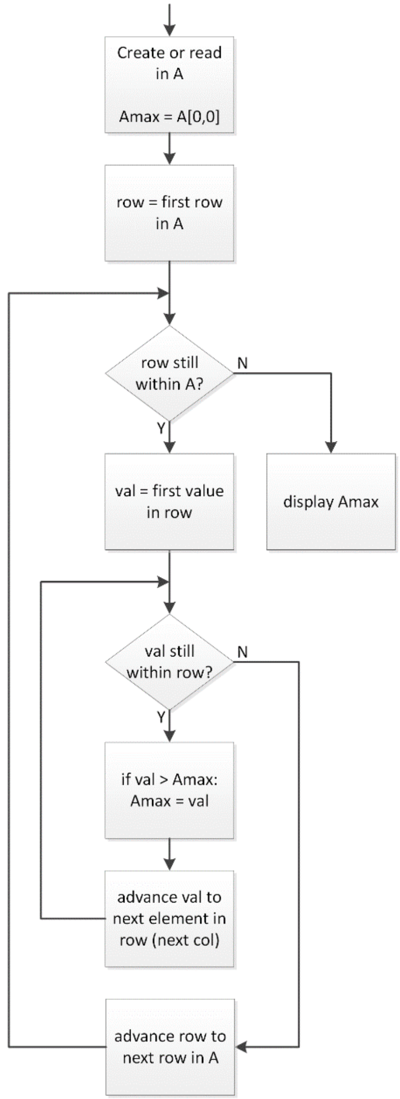
    - val = row[0], val = row[1], and so on



Nested Loop – Example 1


```python

# initialize rng
rng = np.random.default_rng()

# create an array of random integers
A = rng.integers(low=0, high=100, size=(5,5))

# %% find maximum value in A
# initialize Amax
Amax = A[0,0]

# nested for loop to find max value of A
for row in A:
    for val in row:
        if val > Amax:
            Amax = val

print('\n', A)
print('\nAmax = {}'.format(Amax))
```

```
 [[47 95 54 61 66]
 [ 2 20 32 30 91]
 [35  1 60 83 73]
 [29 89 18 94 81]
 [95 53  5 67 90]]

Amax = 95

In [157]:
```



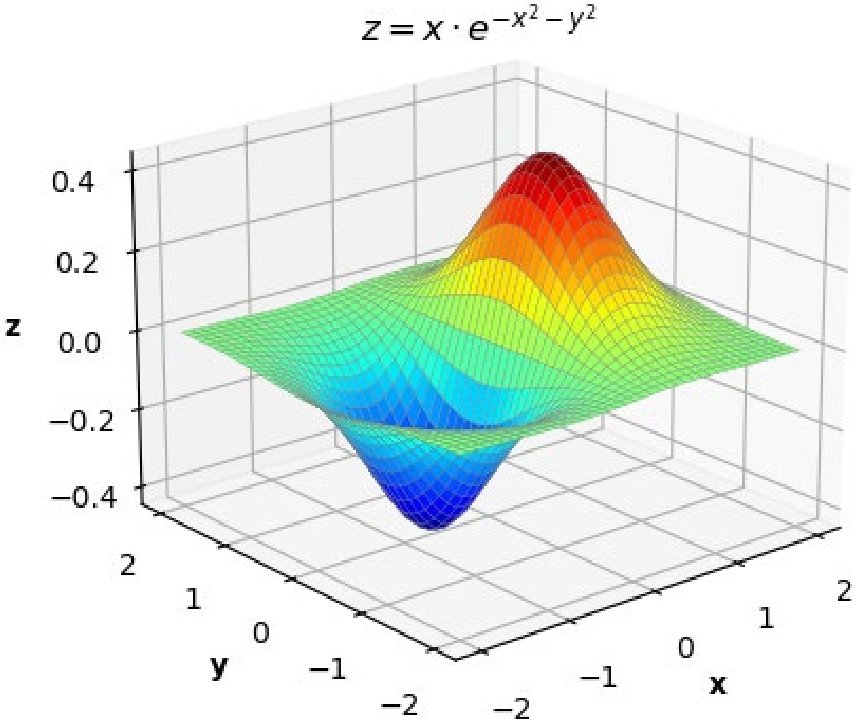
Nested for Loop – Example 2

- Evaluate a function of two variables:

$$z = x \cdot e^{-x^2-y^2}$$

over a range of $-2 \le x \le 2$ and $-2 \le y \le 2$

- A surface in three-dimensional space
- Later in the course, we'll learn how to generate such a plot



Nested for Loop – Example 2

$$z = x \cdot e^{-x^2 - y^2}$$

- Evaluate the function over a range of x and y
- First, define x and y vectors
- Initialize the Z matrix
- Use a nested for loop to step through all points in this range of the x-y plane
  - Use `enumerate()` to iterate through indices and values

```python

x = np.arange(-2, 2.1, 0.1)
y = np.arange(-2, 2.1, 0.1)

Z = np.empty((len(y), len(x)))

for j, xval in enumerate(x):
    for i, yval in enumerate(y):
        Z[i,j] = xval*np.exp(-xval**2 - yval**2)


```

Nested Loops

- We just saw how we can use nested loops to:
  - Find the maximum value in a matrix or 2-D array
  - Evaluate a function of two variables
- A good illustration of nested loops, ***BUT***
- ***There are easier, more efficient ways to do both of these things in Python***
  - Looping is slow – avoid if possible
  - Operate directly on arrays

```
# %%  a better way to evaluate a 2-D
# funtion over a region of the x-y plane

x = np.arange(-2, 2.1, 0.1)
y = np.arange(-2, 2.1, 0.1)

X, Y = np.meshgrid(x,y)

Z = X*np.exp(-X**2 - Y**2)
```

```
# %% a better way to find the maximum
# value in a 2-D array

A = rng.integers(low=0, high=100, size=(5,5))

Amax = np.max(A)

print('\n', A)
print('\nAmax = {}'.format(Amax))
```

33

The Spyder Debugger

Debugging

- You've probably already realized that it's not uncommon for your code to have errors
  - Computer code errors referred to as ***bugs***
- Three main categories of errors
  - ***Syntax errors*** prevent your code from running and generate a Python error message
  - ***Runtime errors*** – not syntactically incorrect, but generate an error upon execution – e.g., indexing beyond matrix dimensions
  - ***Algorithmic errors*** don't prevent your code from executing, but do produce an unintended result
- Syntax and runtime errors are usually more easily fixed than algorithmic errors
- ***Debugging*** – the process of identifying and fixing errors is an important skill to develop
  - Spyder has a built-in ***debugger*** to facilitate this process

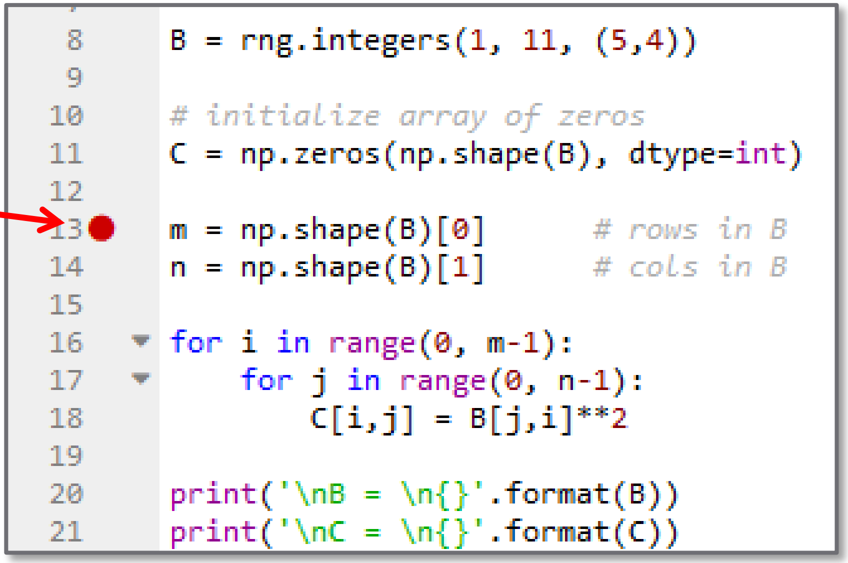
Debugging

- Identifying and fixing errors is difficult because:
  - Programs run seemingly instantaneously
  - Incorrect output results, but can’t see the intermediate steps that produced that output
- ***Basic debugging principles***:
  - ***Slow code execution down*** – allow for stepping through line-by-line
  - ***Provide visibility into the code execution*** – allow for monitoring of intermediate steps and variable values

Spyder Debugger – Breakpoints

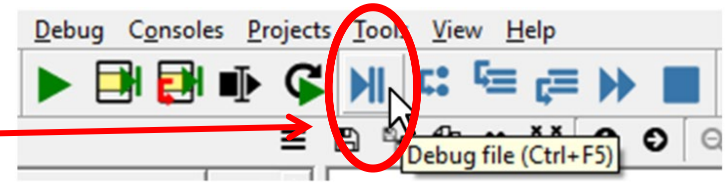
- ***Breakpoint*** – specification of a line of code at which Spyder should pause execution
- Set by clicking next to the number to the left of a line of code in a script
  - Spyder will execute the script ***up to*** this line, then pause

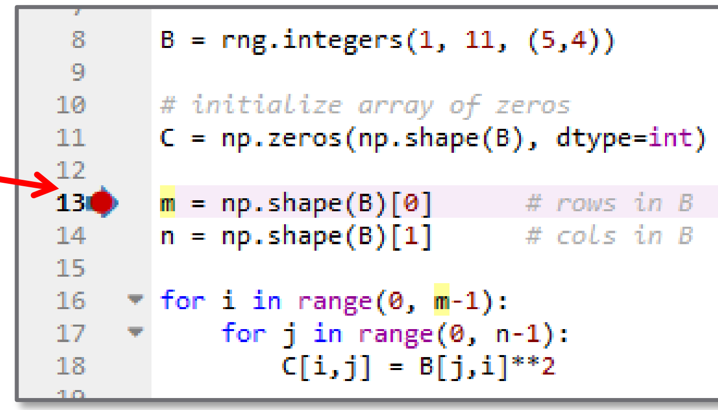
- Clicking here sets a breakpoint
  - Indicated by red circle

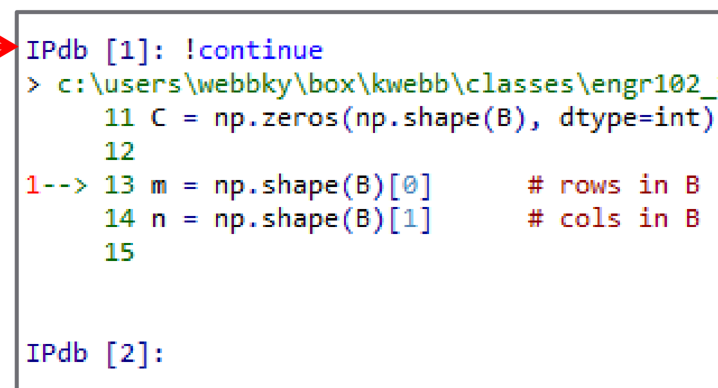


Spyder Debugger – Breakpoints

- Click 'Debug file' to begin execution
- Execution halts at the breakpoint
  - Before executing that line
- Console prompt changes to `IPdb [n]:`
  - Can now interactively enter commands

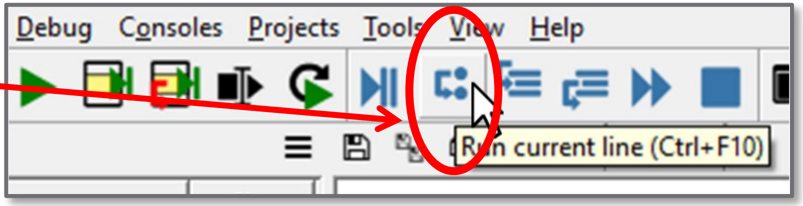




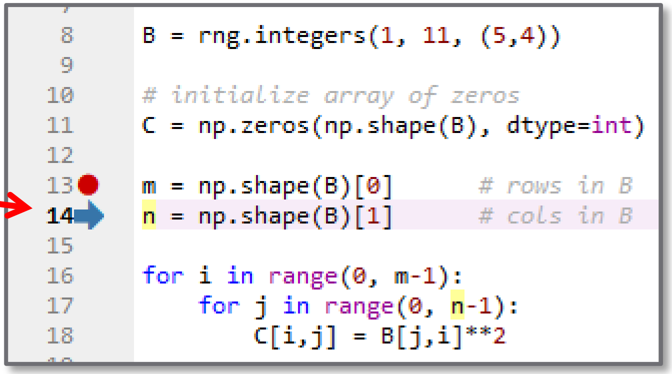


Spyder Debugger – Breakpoints

- Click 'Run current line' to execute the current line of code



- Arrow indicator advances to the next line



- Variable, m, defined on previous line (line 16) now exists in the namespace
  - Available in the console

Debugger – Example

- Recall a previous example of an algorithm to square every element in a matrix
- Let’s say we run our script and get the following result:

```python
# define a matrix of random ints
rng = np.random.default_rng()

B = rng.integers(1, 11, (5,4))

# initialize array of zeros
C = np.zeros(np.shape(B), dtype=int)

m = np.shape(B)[0]        # rows in B
n = np.shape(B)[1]        # cols in B

for i in range(0, m-1):
    for j in range(0, n-1):
        C[i,j] = B[j,i]**2

print('\nB = \n{}'.format(B))
print('\nC = \n{}'.format(C))
```

```
B =
[[ 1  6  1  7]
 [ 3  6  4  7]
 [ 8  6  3  2]
 [ 4  9  1 10]
 [ 4  9  7  7]]

C =
[[ 1  9 64  0]
 [36 36 36  0]
 [ 1 16  9  0]
 [49 49  4  0]
 [ 0  0  0  0]]

In [149]:
```

- Resulting matrix is ***transposed***
  - Use the ***debugger*** to figure out why

Debugger – Example

- Set a ***breakpoint*** in the innermost `for` loop
- Click '***Debug file***'
- Code executes up to the breakpoint
- Variable Explorer shows `i=0` and `j=0`
- Click '***Run current line***'
- Display `B[i,j]` and `C[i,j]` in the console
  - Both are as expected

```
m = np.shape(B)[0]        # rows in B
n = np.shape(B)[1]        # cols in B

for i in range(0, m-1):
    for j in range(0, n-1):
        C[i,j] = B[j,i]**2
```

Name	Type	Size	Value
B	Array of int64	(5, 4)	[[7 3 1 8] [2 6 7 3]
C	Array of int32	(5, 4)	[[0 0 0 0] [0 0 0 0]
i	int	1	0
j	int	1	0
m	int	1	5

```
IPdb [4]: !next

IPdb [4]: B[i,j]
7

IPdb [5]: C[i,j]
49

IPdb [6]:
```

Debugger – Example

- Click '***Run current line***' twice
  - Execute the next iteration of the loop
- Now, `i=0` and `j=1`
  - First row, second column
- `B[i,j] = 10`
- But, `C[i,j] = 16`
  - Should be `100`

Name	Type	Size	Value
B	Array of int64	(5, 4)	[[8 10 6 8] [4 9 6 7]
C	Array of int32	(5, 4)	[[64 16 0 0] [0 0 0 0]
i	int	1	0
j	int	1	1

```
IPdb [2]: !next

IPdb [2]: B[i,j]
10

IPdb [3]: C[i,j]
16

IPdb [4]:
```

Debugger – Example

- We see that C[1,2] = 16 = 4**2 = B[2,1]**2
- This leads us to an error on line 21 of the code
  - Should be B[i,j]**2, not B[j,i]**2

```
 7
 8    B = rng.integers(1, 11, (5,4))
 9
10    # initialize array of zeros
11    C = np.zeros(np.shape(B), dtype=int)
12
13    m = np.shape(B)[0]        # rows in B
14    n = np.shape(B)[1]        # cols in B
15
16    for i in range(0, m-1):
17        for j in range(0, n-1):
18            C[i,j] = B[j,i]**2
19
20    print('\nB = \n{}'.format(B))
21    print('\nC = \n{}'.format(C))
22
23
```

```
IPdb [4]: !continue

B =
[[ 8 10  6  8]
 [ 4  9  6  7]
 [10  3  2  5]
 [ 7 10 10  1]
 [ 6  9  5  8]]

C =
[[ 64  16 100   0]
 [100  81   9   0]
 [ 36  36   4   0]
 [ 64  49  25   0]
 [  0   0   0   0]]

In [151]:
```

Exercise – Nested Loops, Debugger

Exercise

- Write a script to do the following:
  - Create a 5x5 matrix of zeros, X
  - Initialize a random number generator:

    ```
    rng = np.random.default_rng()
    ```
  - In a ***nested loop*** step through all elements in X
    - Outer loop steps through rows, inner loop steps through columns
    - Replace each element in X with a random integer:

      ```
      X[i,j] = rng.integers(100)
      ```
- Set a ***breakpoint*** at the start of the outer loop and run the ***debugger***
- Step through code line-by-line observing the evolution of the matrix X