ISYS90088
Introduction to Application Development

Department of Computing and Information Systems
University of Melbourne

Semester 2 , 2018

Dr Antonette Mendoza

Recap - Selection: if and if-else Statements

- **Selection statements** allow a computer to make choices - based on a **condition**

- The **if** statement is used to create a decision structure, which allows a program to have more than one path of execution.

- The **if** statement causes one or more statements to execute only when a Boolean expression is true.

- It is **a control structure** – a logical design that controls the order in which a set of statements execute.

Recap - The **if – else** statement

- The simplest is a **one-way selection** statement (if)

- Also called a **two-way selection** statement (if-else)

- The condition in the if-else statement must be a Boolean expression – that is, an expression that evaluates to either true or false

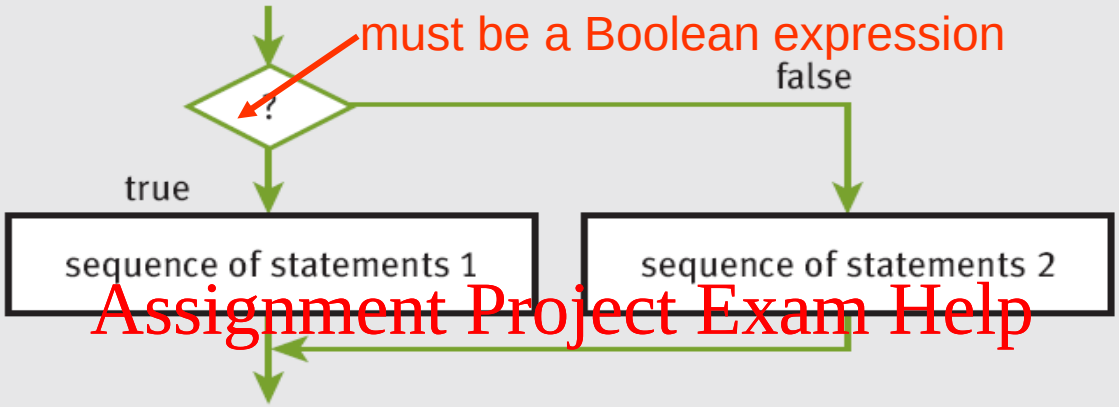
Recap - if-else Statements

- The two possible actions each consist of a sequence of statements
- Each sequence must be indented at least one space beyond the symbols if and else.

Syntax:

```
if <condition>:
    <sequence of statements-1>
else:
    <sequence of statements-2>
```

Recap - if-else Statements (continued)




```
first = int(input("Enter the first number: "))
second = int(input("Enter the second number: "))
if first > second:
    maximum = first
    minimum = second
else:
    maximum = second
    minimum = first
print("Maximum:", maximum)
print("Minimum:", minimum)
```

Multi-way if Statements (continued)

- At most one of the indented blocks will run.
- The conditions are tried in order until one is found that is True. The associated block of code is run and any remaining conditions and blocks are skipped.
- If none of the conditions are True but there is an else block, then Python runs the else block.

Syntax:

```
if <condition-1>:
    <sequence of statements-1>

elif <condition-n>:
    <sequence of statements-n>
else:
    <default sequence of statements>
```

Logical Operators and Compound Boolean Expressions with if-else

- Often a course of action must be taken if either of two conditions is true: Below are two approaches


```
number = int(input("Enter the numeric grade: "))
if number > 100:
    print("Error: grade must be between 100 and 0")
elif number < 0:
    print("Error: grade must be between 100 and 0")
else:
    # The code to compute and print the result goes here
```



```
number = int(input("Enter the numeric grade: "))
if number > 100 or number < 0:
    print("Error: grade must be between 100 and 0")
else:
    # The code to compute and print the result goes here
```

Example: Multi-way if Statements (continued)

Write a program to convert numeric grades of your ISYS90088 test scores to letter grades. The grades are as follows:

- H1 all grades 80 and above(80 - 100)
- H2A all grades equal to and above 75 but below 80 (75-79)
- H2B all grades between grades(70-74)
- P all grades between (50 -69)
- F all grades below 50

Example (using Idle)

Strings manipulations!

- Access individual characters in a string
- Retrieve a substring from a string
- Search for a substring in a string
- Using library for string manipulations
- Splitting a string into lists that can be manipulated

Accessing Characters and Substrings in Strings

- In this section, we examine the internal structure of a string more closely
- You will learn how to extract portions of a string called **substrings**

The Structure of Strings

- **Data structure** – it is an organized way of storing and representing data so that data can be inserted and accessed - data structure consists of smaller pieces of data.

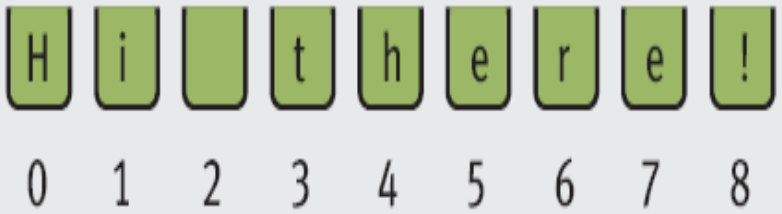
- A string is a data structure.

- The string is a sequence of zero or more characters

- A string is an **immutable** data structure. That means, its internal data elements, the characters, can be accessed, but the structure itself cannot be modified.

The Structure of Strings

- We sometimes might want to access or inspect characters at a given position without needing to visit the entire string (part of a string – substring)
- The string is arranged as shown. There is **an index** that helps us step thru' or inspect a position



Indexing and the Subscript Operator

```
<a string>[<an integer expression>]
```

index is usually in range [0,length of string – 1];
can be negative

- Examples:


```
>>> name = "Alan Turing"
>>> name[0]                          # Examine the first character
'A'
>>> name[3]                          # Examine the fourth character
'n'
>>> name[len(name)]                  # Oops! An index error!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> name[len(name) - 1]              # Examine the last character
'g'
>>> name[-1]                         # Shorthand for the last one
'g'
```

Why?

Strings indexing

- For example: String's length - Number of characters it contains (0+). **len** is a library function that allows us to do some manipulation with strings.

Example:

>>>s = "example"

>>>s[2]

'a'


```
>>> len("Hi there!")
9
>>> len("")
0
```

Negative Indexing

- We have seen what happens when the index is too large. What happens if the index is less than 0? Does it give us an error? NO

```
>>>s = "The number is 42."
>>>print(s[-1])
>>>print(s[-2])
>>>print(s[-3])
>>>print(s[-17])
```


- Negative indices work from the end of the string, so **-1** indexes the last character, which is **??**. and **-17** indexes the 17th last character, which is *T* (actually the first character of the string).
- Note the negative indexes are one-offset (i.e. start from **-1**) while the positive indexes are zero-offset (i.e. start from **0**).

Slicing for Substrings

- Python's subscript operator can be used to obtain a substring through a process called **slicing**
  - Place a colon (**:**) in the subscript; an integer value can appear on either side of the colon
- For example, the following code accesses the substring of the string **"the example is on slicing"** starting at index **2**, up to (but not including) index **10**.

```
>>>s = "the example is on slicing"
>>>print (s[2:10])
e exampl
```

Accessing Substrings (Slicing)

- The notation **2:10** is known as a slice.
- Remember that a slice starts at the first index but finishes one before the end index. This is consistent with indexing: indexing also starts from zero and goes up to one before the length of the string.
- You can see this by slicing with the value of **len**:

```
>>>s = "the example is on slicing"
>>print (len(s))
>>print (s[0:len(s)])
```

More on Slicing: example

- You can also slice with negative indices. The same basic rule of starting from the start index and stopping one before the end index applies:


```
>>>s = ”testing slicing 101"
>>>print(s[4:-7])
>>>print(s[-7:-1])
>>>print(s[-6:len(s)])
```


Solution:
????

Example - check example using IDLE

More on slicing: example

- Python provides two shortcuts for common slice values:
  - if the start index is 0 then you can leave it blank
  - if the end index is the length of the string then you can leave it blank


```
>>>s = ”testing slicing 101”
>>>print(s[:5])
>>>print(s[5:])
>>>print(s[:])
```

Solution:???

Changing the Step Size and Direction

- You can specify a third number which indicates how much to step through the list by.
- For example, if you want every second element you can do this:

```
>>>s = "abcdef"
>>>print(s[::2])
```


- Or you can specify a range with a step:

```
>>>s = "abcdef"
>>>print(s[0:3:2])
```

Example – show using IDLE!

Changing the Step Size and Direction

- If this third number is -1 it changes the direction you are slicing in:

```
>>>s = "abcdef"
>>>print(s[2::-1])
>>>print(s[2:0:-1])
>>>print(s[-4:-6:-1])
cba
cb
cb
```


- Note that the direction of the indices must be changed also. If there is nothing in between the indices, an empty string is returned (unlike indexing beyond the ends of the string, which led to an error):

```
>>>s = "abcdef"
>>>print(s[0:2:-1])
```

String Methods (continued)

- Methods can expect arguments and return values
- A method knows about the internal state of the object with which it is called
- In Python, all data values are objects
- dir(str) or help(str) will give you the entire list of string methods or library functions.

String Methods (continued)

STRING METHOD	WHAT IT DOES
`s.center(width)`	Returns a copy of `s` centered within the given number of columns.
`s.count(sub [, start [, end]])`	Returns the number of non-overlapping occurrences of substring `sub` in `s`. Optional arguments `start` and `end` are interpreted as in slice notation.
`s.endswith(sub)`	Returns `True` if `s` ends with `sub` or `False` otherwise.
`s.find(sub [, start [, end]])`	Returns the lowest index in `s` where substring `sub` is found. Optional arguments `start` and `end` are interpreted as in slice notation.
`s.isalpha()`	Returns `True` if `s` contains only letters or `False` otherwise.
`s.isdigit()`	Returns `True` if `s` contains only digits or `False` otherwise.

String Methods (continued)

STRING METHOD	WHAT IT DOES
`s.join(sequence)`	Returns a string that is the concatenation of the strings in the sequence. The separator between elements is `s`.
`s.lower()`	Returns a copy of `s` converted to lowercase.
`s.replace(old, new [, count])`	Returns a copy of `s` with all occurrences of substring `old` replaced by `new`. If the optional argument `count` is given, only the first `count` occurrences are replaced.
`s.split([sep])`	Returns a list of the words in `s`, using `sep` as the delimiter string. If `sep` is not specified, any whitespace string is a separator.
`s.startswith(sub)`	Returns `True` if `s` starts with `sub` or `False` otherwise.
`s.strip([aString])`	Returns a copy of `s` with leading and trailing whitespace (tabs, spaces, newlines) removed. If `aString` is given, remove characters in `aString` instead.
`s.upper()`	Returns a copy of `s` converted to uppercase.

String Methods (continued)

```
>>> s = "Hi there!"
>>> len(s)
9
>>> s.center(11)
' Hi there! '
>>> s.count('e')
2
>>> s.endswith("there!")
True
>>> s.startswith("Hi")
True
>>> s.find('the')
3
>>> s.isalpha()
False
>>> 'abc'.isalpha()
True
>>> "326".isdigit()
True
>>> words = s.split()
>>> words
['Hi', 'there!']
>>> "".join(words)
'Hithere!'
```

String Methods (continued)

```
>>> " ".join(words)
'Hi there!'
>>> s.lower()
'hi there!'
>>> s.upper()
'HI THERE!'
>>> s.replace('i', 'o')
'Ho there!'
>>> " Hi there! ".strip()
'Hi there!'
>>>
```

list method

- The method **list()** takes sequence types and converts them to lists.

Syntax:

list(seq)

```
>>> num = '123'
>>> list(num)
['1', '2', '3']
```

Testing for substring with the **in** Operator

- When used with strings, the left operand of **in** is a target substring and the right operand is the string to be searched
  - Returns **True** if target string is somewhere in search string, or **False** otherwise

- Example:

```
>>>list = [“ant”, “dog”, “cat”, “rat”, “horse”]
>>>”ant” in list
True
```

(run example code for counting vowels)

An example for split

Example:

```
>>>S= ‘this is an example’
>>>listofitems = s.split()
>>>print(listofitems)

example(1)
```

Example : program – try this one!

Imagine you are writing a title for a business report in your organization. The title should be not more that 10 words. Write a program that accepts from the user the title and :

i. Makes sure the title is in upper case. If its not, your program converts the title to all upper case;

i. counts the number of words in the title. If the number of words is >10, your program will inform the user that the title must be not more that 10 words.

Example : fill in the XXX !

```
text = input("enter a sentence :")
If XXX:
    text = text.upper()
print('your title in uppercase is', text)
listofwords =XXX
if XXX > 10:
    print("your title must not exceed 10 words")
else:
    print ("there are", XXX, "words in this title")
```

Example : fill in the XXX !

```
text = input("enter a sentence :")
If text.islower():
    text = text.upper()
print('your title in uppercase is', text)
listofwords =text.split()
If listofwords > 10:
    print("your title must not exceed 10 words")
else:
    print ("there are", len(listofwords), "words in this title")
```

Break

Loops in Python

- Python programming language provides following types of loops to handle looping requirements.
- **Why do we need loops?**

- Types of loops:

  - **for loop:** Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

  - **while loop:** Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.

  - **nested loops:** can use one or more loop inside any another while, for or do..while loop.

The **for** Loop

- Repetition statements (or **loops**) repeat an action
- Each repetition of action is known as **pass** or **iteration**
- Python's **for** loop is the control statement that supports definite iteration
- A **for** loop helps with control, counting and repetition

The **for** loop

- It has the ability to iterate over the items of any sequence, such as **a list** or **a string**.

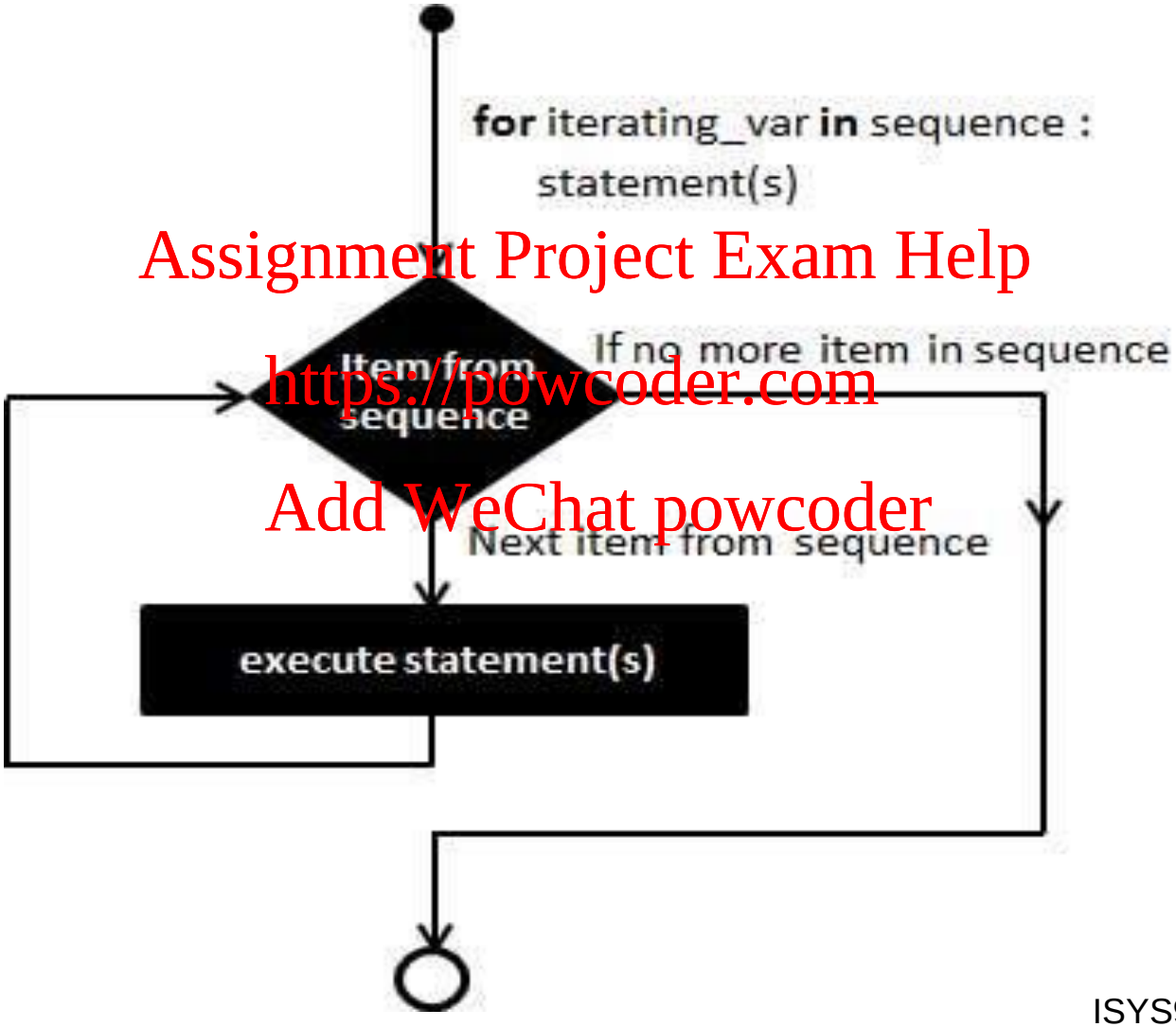
Syntax:

for *iterating_var* **in** sequence:

<statements(s)>

- If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *iterating_var*. Next, the statements block is executed.
- Each item in the list is assigned to *iterating_var*, and the statement(s) block is executed until the entire sequence is exhausted.

The **for** loop



Examples of **for** loops: using list and strings

```
#example 1 - this is one way of doing it
for num in [1, 2, 3, 4, 5]:
    print (num)

#example 2 - this is another way of doing it using lists
listofitems = [1, 2, 3, 4, 5];
for num in listofitems:
    print (num)

# Example using strings
for letter in 'Python':
  print ('Current Letter :', letter)

# Second Example using strings
fruits = ['banana', 'apple',  'mango']
for fruit in fruits:
   print ('Current fruit :', fruit)
```

Traversing the Contents of a Data Sequence

- Strings are also sequences of characters
- Values in a sequence can be visited with a **`for`** loop:

- Example: **character** is called the target variable- that takes a value of each loop iteration


```
for <variable> in <sequence>:
   <do something with variable>
```

```
>>> for character in "Hi there!":
        print(character, end = " ")

H i   t h e r e !
>>>
```

Executing a Statement a Given Number of Times using the **range** function

```
>>> for eachPass in range(4):
        print("It's alive!", end=" ")

It's alive! It's alive! It's alive! It's alive!
>>>
```


- The form of this type of loop is:

```
for <variable> in range(<an integer expression>):
    <statement-1>
    ...
    <statement-n>
```

loop header

loop body

statements in body must be indented and aligned in the same column

Traversing the Contents of a Data Sequence

- **`range`** returns a **list**


```
>>> list(range(4))
[0, 1, 2, 3]
>>> list(range(1, 5))
[1, 2, 3, 4]
>>>
```

Executing a Statement a Given Number of Times (continued)

- Example: Loop to compute an exponentiation for a non-negative exponent


```
>>> number = 2
>>> exponent = 3
>>> product = 1
>>> for eachPass in range(exponent):
        product = product * number
        print(product, end = " ")

2 4 8
>>> product
8
```


- The variable **product** is called an accumulator
- If the exponent were 0, the loop body would not execute and value of **product** would remain as 1

Count-Controlled Loops

- Loops that count through a range of numbers

```
>>> product = 1
>>> for count in range(4):
        product = product * (count + 1)

>>> product
24
```


- To specify an explicit lower bound:

```
>>>product = 1
>>>for count in range(1:5):
        product = product * count

>>>product
24
```

Count-Controlled Loops (continued)

- Example: bound-delimited **summation**


```
>>> lower = int(input("Enter the lower bound: "))
Enter the lower bound: 1
>>> upper = int(input("Enter the upper bound: "))
Enter the upper bound: 10
>>> sum = 0
>>> for count in range(lower, upper + 1):
        sum = sum + count

>>> sum
55
>>>
```

Note: Augmented Assignment

- **Augmented assignment operations**:

```
a = 17
s = "hi"

a += 3          # Equivalent to a = a + 3
a -= 3          # Equivalent to a = a - 3
a *= 3          # Equivalent to a = a * 3
a /= 3          # Equivalent to a = a / 3
a %= 3          # Equivalent to a = a % 3
s += " there"   # Equivalent to s = s + " there"
```


- Format:

  Equivalent to:

```
<variable> <operator>= <expression>
```

```
<variable> = <variable> <operator> <expression>
```

Loop Errors: Off-by-One Error

- Example – if you want to count from 1 to 4???????:

```
for count in range(1, 4):    # Count from 1 through 4, we think
    print(count)
```


Loop actually counts from 1 through 3

- This is not a syntax error, but rather a logic error

Specifying the Steps in the Range

- **range** expects a third argument that allows you specify a **step value**

```
>>> list(range(1, 6, 1))    # Same as using two arguments
[1, 2, 3, 4, 5]
>>> list(range(1, 6, 2))    # Use every other number
[1, 3, 5]
>>> list(range(1, 6, 3))    # Use every third number
[1, 4]
>>>
```


- Example in a loop:

```
>>> sum = 0
>>> for count in range(2, 11, 2):
        sum += count

>>> sum
30
>>>
```

Loops That Count Down

- Example:


```
>>> for count in range(10, 0, -1):
        print(count, end=" ")

10 9 8 7 6 5 4 3 2 1
>>> list(range(10, 0, -1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Quiz

1. Write the output of the following loops:

   a. **for** count **in** range(5)
   print(count + 1, end = " ")

   b. **for** count **in** range(1, 4):
   print(count)

   c. **for** count **in** range(1, 6, 2):
   print(count)

   d. **for** count **in** range(6, 1, -1):
   print(count)

Example

```
#write a program in python that displays numbers starting from
#through to a user's requirement and their squares in a table
#student ex: fill the places marked  XXX with accurate
#statements that satisfies the requirement of the question
```



```
start = xxx
end = xxx
print ('Number\t Square')
print('--------------')
for num in range (xxx, xxx):
    square = num **2
    print(num, '\t', square)
```

- Break (if time permits continue)

Conditional Iteration: The `while` Loop

- The **while** loop can be used to describe conditional iteration
  - Example: A program's input loop that accepts values until user enters a '**sentinel**' that terminates the input

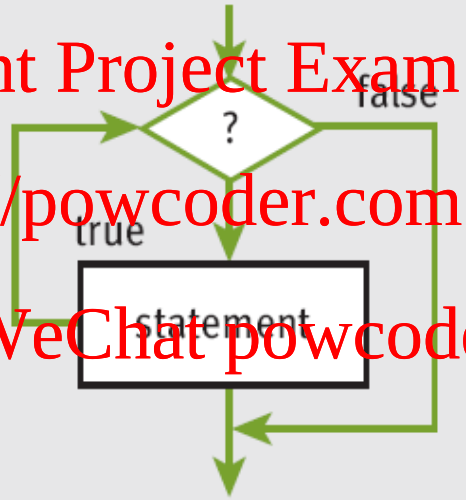
The Structure and Behavior of a `while` Loop

- Conditional iteration requires that condition be tested within loop to determine if it should continue
  - Called **continuation condition**


```
while <condition>:
    <sequence of statements>
```


  - Improper use may lead to **infinite loop**
- **while** loop is also called **entry-control loop**
  - Condition is tested at top of loop
  - Statements within loop can execute zero or more times

The Structure and Behavior of a `while` Loop (continued)



The Structure and Behavior of a `while` Loop (continued)

```
sum = 0.0
data = input("Enter a number or just enter to quit: ")
while data != "":
    number = float(data)
    sum += number
    data = input("Enter a number or just enter to quit: ")
print("The sum is", sum)

Enter a number or just enter to quit: 3
Enter a number or just enter to quit: 4
Enter a number or just enter to quit: 5
Enter a number or just enter to quit:
The sum is 12.0
```


data is the **loop control variable**

Count Control with a `while` Loop

```
sum = 0
for count in range(1, 100001):
    sum += count
print(sum)
```

```
sum = 0
count = 1
while count <= 100000:
    sum += count
    count += 1
print(sum)
```



```
for count in range(10, 0, -1):
    print(count, end=" ")

count = 10
while count >= 1:
    print(count, end=" ")
    count -= 1
```

The `while True` Loop and the `break` Statement

- **while** loop can be complicated to write correctly
  - Possible to simplify its structure and improve its readability


```
sum = 0.0
while True:    ← a simple way to create a controlled exit
    data = input("Enter a number or just enter to quit: ")
    if data == "":    ← loop's termination condition
        break    ← causes an exit from the loop
    number = float(data)
    sum += number
print("The sum is", sum)
```

The `while True` Loop and the `break` Statement (continued)

```
while True:
    number = int(input("Enter the numeric grade: "))
    if number >= 0 and number <= 100:
        break
    else:
        print("Error: grade must be between 100 and 0")
print(number)    # Just echo the valid input
```


- Alternative: Use a Boolean variable to control loop


```
done = False
while not done:
    number = int(input("Enter the numeric grade: "))
    if number >= 0 and number <= 100:
        done = True
    else:
        print("Error: grade must be between 100 and 0")
print(number)    # Just echo the valid input
```

break statement

- It terminates the current loop and resumes execution at the next statement, just like the traditional break statement in C.

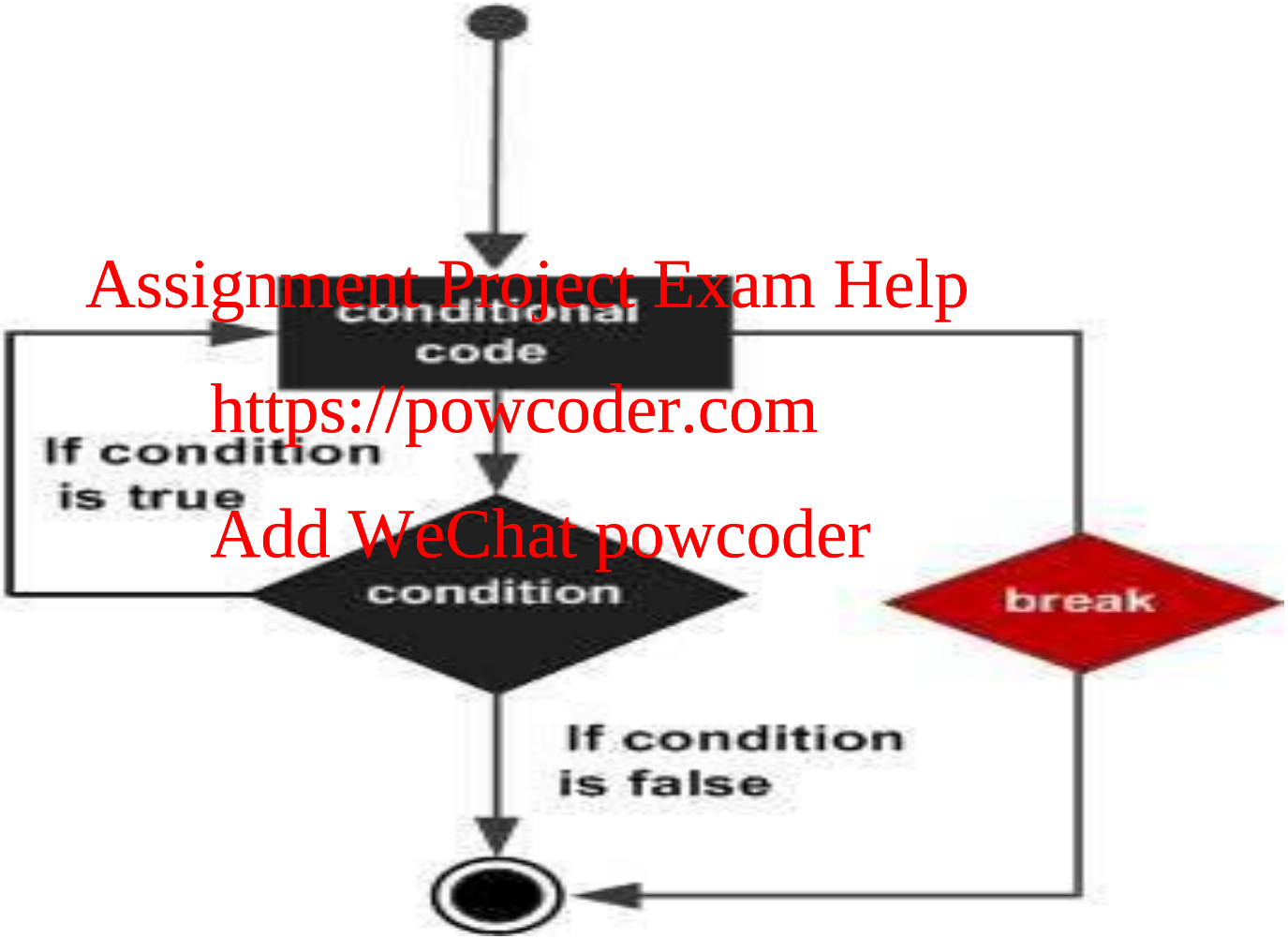
- The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The break statement can be used in both while and for loops.

- If you are using nested loops, the break statement stops the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

```
>>> break
```

break statement



Example