New Topic

Repetition and Loops

Additional Python constructs that allow us to effect the (1) order and (2) **number of times** that program statements are executes.

These constructs are the

1. **while loop** and
2. **for loop**.

Recall:

1. Input
2. Output
3. Memory
4. Arithmetic – symbolic manipulation and evaluation
5. Control – we have already seen "straight line execution, and if statement.

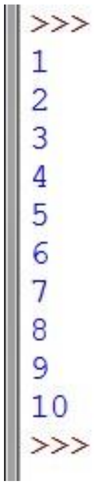
We have seen a simple example of control. Python simply executes one statement after another from the first statement of the program to the last i.e. straight line execution. In we have seen how Python lets us execute statements **conditionally**. So if a Boolean condition is met, a given block of statements is executed **once**, otherwise it is not.

But … what about the case that we want to execute the block multiple times, as long as the Boolean condition is True? We can do this by using a

while loop.

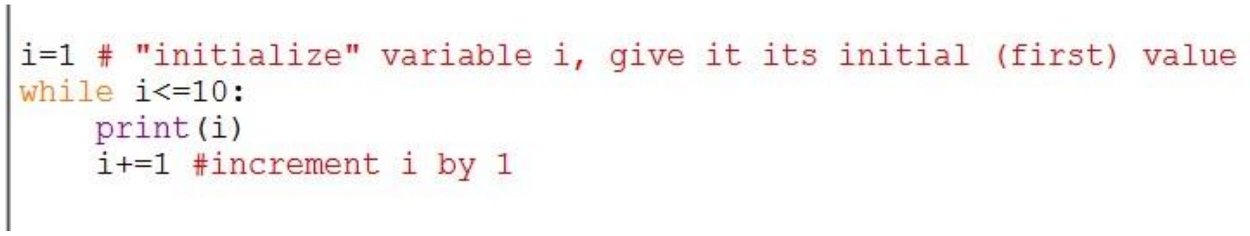
Example:

Write a program that prints the numbers 1 – 10, one per line.



```
>>>
1
2
3
4
5
6
7
8
9
10
>>>
```

Solution: Use a while loop.



```
i=1 # "initialize" variable i, give it its initial (first) value
while i<=10:
    print(i)
    i+=1 #increment i by 1
```

The **syntax** (form) of the while statement is:

while <Boolean expression> :
one or more statements (called a code block)

The **semantics** (meaning or interpretation) is:

Upon encountering the while statement:

1. Evaluate the Boolean expression
2. Repeat the associated block as long as the Boolean expression is True
3. Exit the **while loop** as soon as the Boolean expression evaluates to False.

Definition: This process of repeating one or more statements is called <u>**iteration**</u>.

Problem:

Write a program that asks the user for a positive integer n. It then prints the numbers 1 through n, one per line.

Answer:

Problem:
Write a program that asks the user for a positive integer n. It then prints the numbers **n down to 1,** one per line.

Answer:

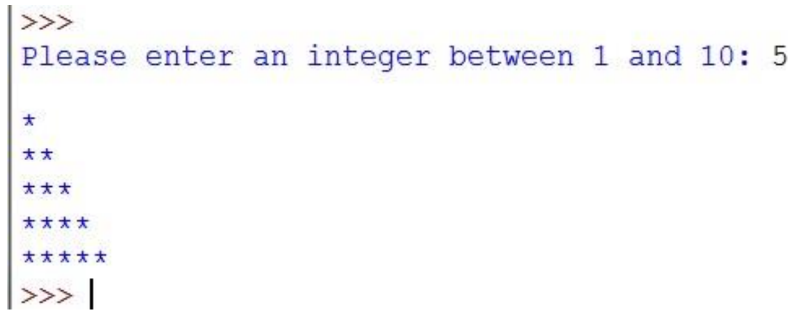
Problem:

Write a program that asks the user for a positive integer n (>= 2). It then prints the even numbers in the range 1 to n, **five** per line.

Answer:

Problem:

Write a program that asks the user for a positive integer n. Print a triangle of “stars”(= “*”) like in the example below.



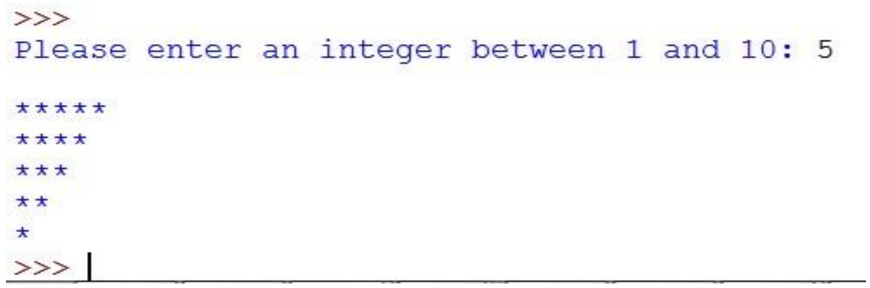
```
>>>
Please enter an integer between 1 and 10: 5

*
**
***
****
*****
>>>
```

Answer:

Problem:

Now do this one.



```
>>>
Please enter an integer between 1 and 10: 5

*****
****
***
**
*
>>>
```

Problem:

Write a program that asks the user for a positive integer n. Calculate and print the **sum** of the integers 1 through n.

Answer:

Problem:

Write a program that asks the user for a positive integer n. Calculate and print the **product** of the integers 1 through n.

Answer:

Problem:

Write a program that asks the user for a positive integer n. Calculate and print the **sum of the odd integers** in the range 1 through n.

Answer:

Problem:

Write a program to **request and validate a password** from your user. A valid password will be any two digit integer, **both digits of which are even**.

Give the user **three** chances to enter a correct password.

- At each incorrect attempt, print “Invalid password. Try again”
- If the password entered is correct print “Correct! You may access the system.” Exit the program.
- If the password entered is incorrect print “Too many invalid attempts. Please try again later.”

Answer:

Problem:

Write a program that asks the user for a positive integer n where the right-most digit is not a zero. Print out the digits of n from right to left – one next to the other. So if the input is 123 your program will print out the digits 3,2,1 next to each othergiving 321.

Answer:

Problem:

Write a program that asks the user for a positive integer n where the right-most digit is not a zero. **Construct and output the integer whose digits are the reverse of those in n.**

For example, if n has the value 123, then you need to construct the integer 321. **Note** you are not just printing the digits of n in reverse order; you are actually constructing the new integer.

Answer:

1. What is the algorithm (method)?

2. Write the code.

Problem:

Recall the definition of a prime number:

An integer greater than one is called a prime number if its only positive divisors (factors) are one and itself.

Write a program that inputs a positive integer n and determines if n is prime or composite (i.e. not a prime).

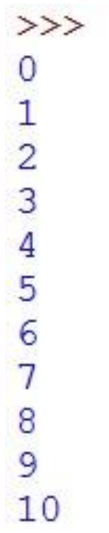
Answer:

We now look at the second loop structure in Python, the

for loop

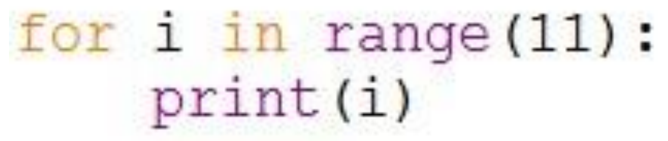
Example:

Write a program that prints the numbers 0 – 10, one per line.



```
>>>
0
1
2
3
4
5
6
7
8
9
10
```

Answer:



```
for i in range(11):
    print(i)
```

This is the simplest usage of the for statement. It lets you iterate over a sequence of numbers. In the example above, the sequence starts implicitly at 0 and goes up to, but not including 11.

The **syntax** (form) of this form of the for statement is:

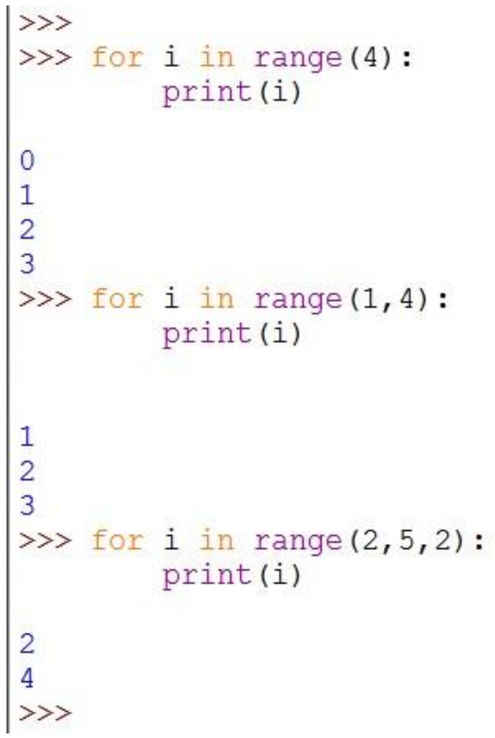
for <some variable> in range(…) :
one or more statements (called a code block)

The **semantics** (meaning or interpretation) is:

- The variable after the "for" takes on each value in the "range" successively.
- That value may be used in the block that is the body of the for statement.

Question: What about **the range function**?

Here are some examples:

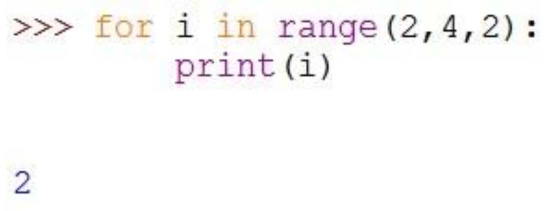

```
>>>
>>> for i in range(4):
        print(i)

0
1
2
3
>>> for i in range(1,4):
        print(i)


1
2
3
>>> for i in range(2,5,2):
        print(i)

2
4
>>>
```

and


```
>>> for i in range(2,4,2):
        print(i)


2
```

range(…) has the following **syntax**:

range([start value,] end value [,step])

and the following **semantics**:

Ranges (from Python help)

The `range` type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in `for` loops.

class `range`(*stop*)
class `range`(*start*, *stop*[, *step*])

The arguments to the range constructor must be integers (either built-in `int` or any object that implements the `__index__` special method). If the *step* argument is omitted, it defaults to `1`. If the *start* argument is omitted, it defaults to `0`. If *step* is zero, `ValueError` is raised.

For a positive *step*, the contents of a range `r` are determined by the formula `r[i] = start + step*i` where `i >= 0` and `r[i] < stop`.

For a negative *step*, the contents of the range are still determined by the formula `r[i] = start + step*i`, but the constraints are `i >= 0` and `r[i] > stop`.

A range object will be empty if `r[0]` does not meet the value constraint. Ranges do support negative indices, but these are interpreted as indexing from the end of the sequence determined by the positive indices.

Ranges containing absolute values larger than `sys.maxsize` are permitted but some features (such as `len()`) may raise `OverflowError`.

Range examples:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

Ranges implement all of the common sequence operations except concatenation and repetition (due to the fact that range objects can only represent sequences that follow a strict pattern and repetition and concatenation will usually violate that pattern).

`start`

The value of the *start* parameter (or `0` if the parameter was not supplied)

`stop`

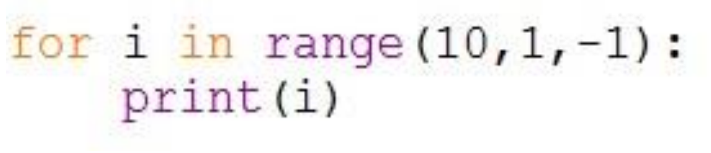
The value of the *stop* parameter

`step`

The value of the *step* parameter (or `1` if the parameter was not supplied)

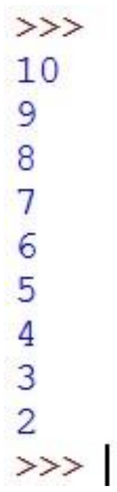
The advantage of the `range` type over a regular `list` or `tuple` is that a `range` object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the `start`, `stop` and `step` values, calculating individual items and subranges as needed).

Example: The following code:



```
for i in range(10,1,-1):
    print(i)
```

prints:



```
>>>
10
9
8
7
6
5
4
3
2
>>> |
```

Use a for loop to solve the next set of problems.

Problem:

Write a program that asks the user for a positive integer n. It then prints the numbers 1 through n, one per line.

Answer:

Problem:

Write a program that asks the user for a positive integer n. Calculate and print the **sum** of the integers 1 through n.

Answer:

Problem:

Write a program that asks the user for a positive integer n. Calculate and print the **product** of the integers 1 through n.

Answer:

Problem:

Write a program that asks the user for a positive integer n. Calculate and print the **sum of the odd integers** in the range 1 through n.

Answer:

Problem:

Generate and print all numbers between 1 and 1000 such that the sum of the digits equals 20.

Answer:

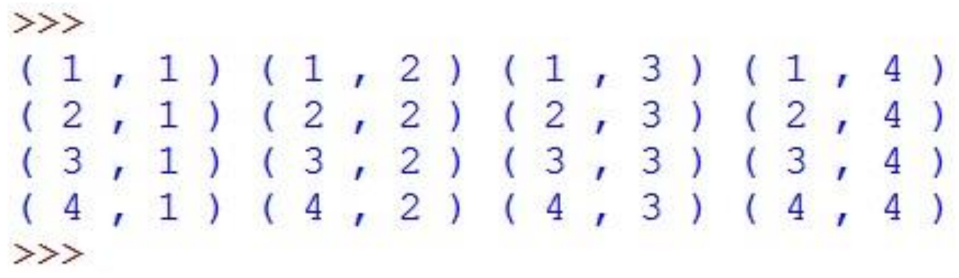
Problem:

Generate and print all prime numbers between 1 and 100.

Answer:

Problem:

Write a program using for loops to print the following:

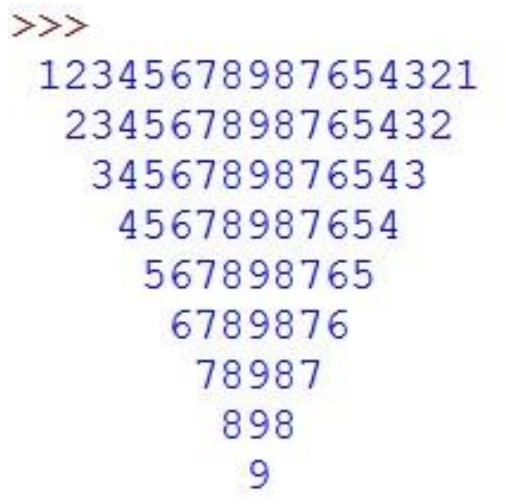


```
>>>
( 1 , 1 ) ( 1 , 2 ) ( 1 , 3 ) ( 1 , 4 )
( 2 , 1 ) ( 2 , 2 ) ( 2 , 3 ) ( 2 , 4 )
( 3 , 1 ) ( 3 , 2 ) ( 3 , 3 ) ( 3 , 4 )
( 4 , 1 ) ( 4 , 2 ) ( 4 , 3 ) ( 4 , 4 )
>>>
```

Answer:

Problem:

Write a program, using for loops, to generate the following output.



```
>>>
 12345678987654321
  234567898765432
   3456789876543
    45678987654
     567898765
      6789876
       78987
        898
         9
```

Answer:

How do we control statement execution inside a loop (either while or for)? Two new ways.

Until now: **while**

loop:

We keep executing the body of the loop as long as the Boolean expression in the loop head is true. We exit only when it becomes false.

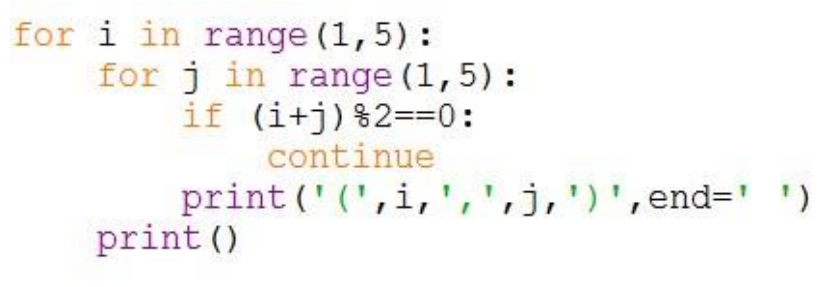
for loop:

We keep executing the body of the loop as long as the sequence of values in the range() function has not been exhausted. We exit only when there are no more values generated in conjunction with the "argument list".

However …. Python has two other statements that allow us to control what happens inside a loop:

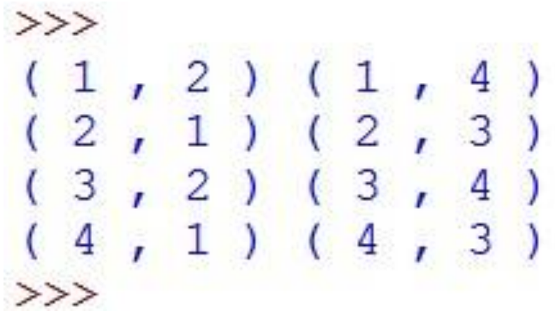
1. **continue**
2. **break**

1. continue: When Python encounters a **continue statement** inside a loop, the interpreter proceeds directly to the top of the loop, skipping all the statements in the after the continue. For example this code



```
for i in range(1,5):
    for j in range(1,5):
        if (i+j)%2==0:
            continue
        print('(',i,',',j,')',end=' ')
    print()
```

produces this output:



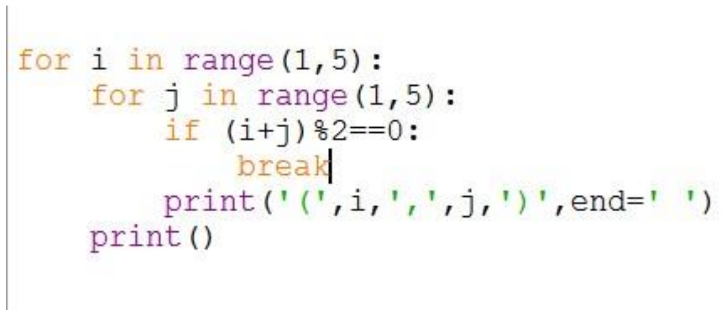
```
>>>
( 1 , 2 ) ( 1 , 4 )
( 2 , 1 ) ( 2 , 3 )
( 3 , 2 ) ( 3 , 4 )
( 4 , 1 ) ( 4 , 3 )
>>>
```

Question: What would be printed if the if and continue are left out?

2. **break:** When Python encounters a **break statement** inside a loop, the interpreter causes the loop to terminate and execution of the program continues with the first statement after the loop.

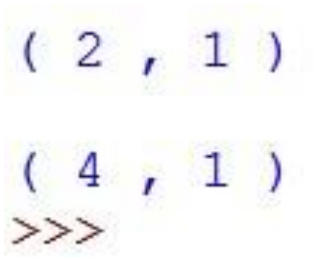
For example this code

For example this code



```
for i in range(1,5):
    for j in range(1,5):
        if (i+j)%2==0:
            break
        print('(',i,',',j,')',end=' ')
    print()
```

produces this output:



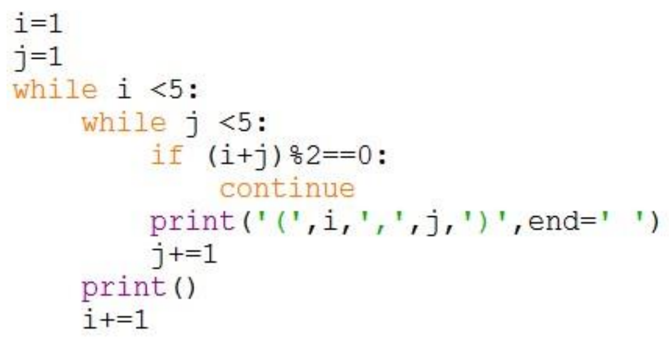
```
( 2 , 1 )

( 4 , 1 )
>>>
```

Question: explain this output in detail, including the blank line between the tuples printed.

Here is an attempt to re-write the "continue" example with the for loop except using a while loop:

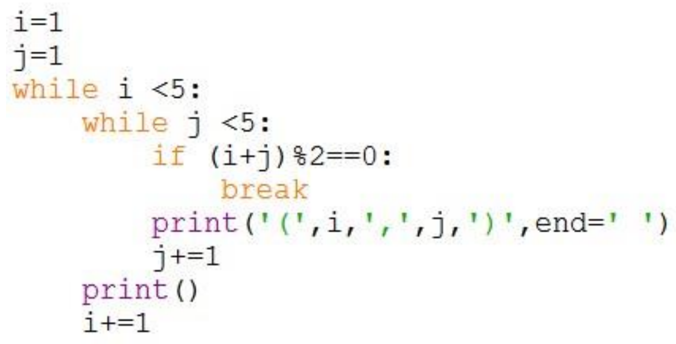
example



```
i=1
j=1
while i <5:
    while j <5:
        if (i+j)%2==0:
            continue
        print('(',i,',',j,')',end=' ')
        j+=1
    print()
    i+=1
```

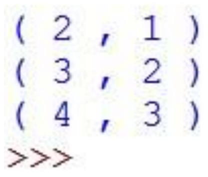
Question: What will be printed? Why?

If instead of a **continue** statement in the code above, we put a **break**, like this:



```
i=1
j=1
while i <5:
    while j <5:
        if (i+j)%2==0:
            break
        print('(',i,',',j,')',end=' ')
        j+=1
    print()
    i+=1
```

we get



```
( 2 , 1 )
( 3 , 2 )
( 4 , 3 )
>>>
```

Why?

The for iterates through any “iterable”, we saw the range function.

Since looping over **ranges of integers** is quite common, there is a shortcut:

```
for n in range(1, 10):
   print(f'2 to the {n} power is {2**n}')
```

The range(i, j [,step]) function creates an object that represents a range of integers with values from i up to, but not including, j. If the starting value is omitted, it’s taken to be zero. An optional stride can also be given as a third argument. Here are some examples:

```
a = range(5)        # a = 0, 1, 2, 3, 4

b = range(1, 8)     # b = 1, 2, 3, 4, 5, 6, 7

c = range(0, 14, 3)  # c = 0, 3, 6, 9, 12

d = range(8, 1, -1)  # d = 8, 7, 6, 5, 4, 3, 2
```

The object created by range() computes the values it represents on demand when lookups are requested. Thus, it’s efficient to use even with a large range of numbers.

The range() function simplifies number generation. Adjust start, stop, and step for flexible sequences, including descending and negative ranges.

Some more detail:

Syntax

range(start, stop, step)

- Beginning of sequence (default: 0).
- End of sequence (exclusive – up to but not including End vslue).
- Increment (default: 1). Can be negative.

Some more examples

1. Basic Usage

```
for i in range(5):
   print(i)  # 0, 1, 2, 3, 4
```

General Syntax: range(stop)
Use case: Quickly generate a sequence from 0 to a given number (exclusive). Useful for iterating a specific number of times in loops.

2. Start and Stop

```
for i in range(2, 7):

    print(i)  # 2, 3, 4, 5, 6
```

General Syntax: range(start, stop)
Use case: Define a custom starting point. Handy when you need to iterate over a subset of a range.

3. Positive Step

```
for i in range(1, 10, 2):

    print(i)  # 1, 3, 5, 7, 9
```

General Syntax: range(start, stop, step)
Use case: Skip numbers in a sequence. Great for processing every nth item or generating spaced values.

4. Negative Step

```
for i in range(5, 0, -1):

    print(i)  # 5, 4, 3, 2, 1
```

General Syntax: range(start, stop, step) with a negative step
Use case: Reverse iteration. Useful for counting down or reversing sequences in loops.

5. Negative Start and Stop

```
for i in range(-5, 0):

    print(i)  # -5, -4, -3, -2, -1
```

General Syntax: range(start, stop) with negative values for start and stop
Use case: Generate negative ranges. Commonly used in algorithms requiring negative indices or offsets.

6. Negative Range with Step

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

    print(i)  # -1, -2, -3, -4, -5
```

General Syntax: range(start, stop, step) with both negative values and a negative step
Use case: Combine negative ranges and steps for flexible sequences, such as reverse iteration over negative values.

Try these:

Print numbers from 10 to 1.

Generate multiples of 4 between -8 and 8.

Create a range from -10 to -1 with a step of 2.

Iterate over range(-3, 4, 2) and print each number.

The for statement is not limited to sequences of integers. It can be used to iterate over many kinds of objects including strings, lists, dictionaries, and files. Here’s an example:

```
for n in [1, 2, 3, 4, 5, 6, 7, 8, 9]:

    print(f'2 to the {n} power is {2**n}')
```

In this example, the variable n will be assigned successive items from the list [1, 2, 3, 4, ..., 9] on each iteration.

```
message = 'Hello World'
# Print out the individual characters in message
for c in message:
    print(c)

names = ['Dave', 'Mark', 'Ann', 'Phil']
# Print out the members of a list
for name in names:
    print(name)



prices = { 'GOOG' : 490.10, 'IBM' : 91.50, 'AAPL' : 123.15 }
# Print out all of the members of a dictionary

for key in prices:
    print(key, '=', prices[key])


# Print all of the lines in a file
with open('foo.txt') as file:
    for line in file:
        print(line, end='')
```

The for loop is one of Python’s most powerful language features because you can create custom iterator objects and generator functions that supply it with sequences of values.