Introduction to Programming in Python

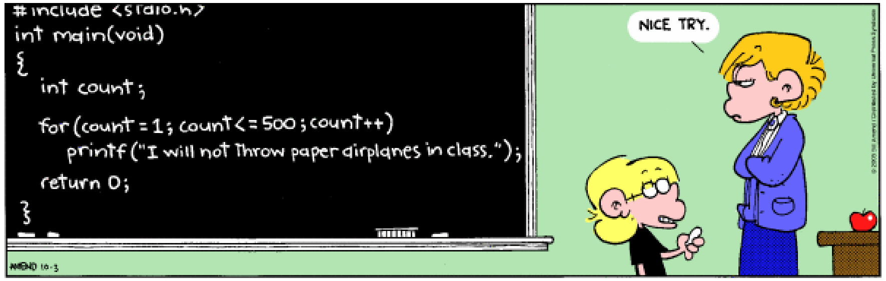
Loops

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: June 4, 2021 at 11:04

Repetitive Activity

Often we need to do some (program) activity numerous times:
So you might as well use cleverness to do it. *That's what loops are for.*



```
#include <stdio.h>
int main(void)
{
  int count;

  for(count=1;count<=500;count++)
     printf("I will not Throw paper airplanes in class.");
  return 0;
}
```

It doesn't have to be the exact same thing over and over.

While Loop

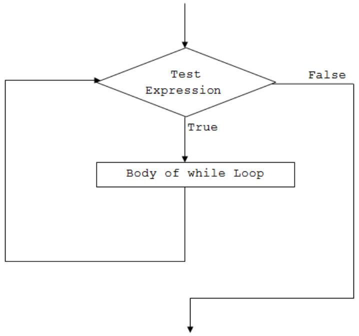
One way is to use a `while` loop. It is typical to use a `while` loop if you don't know exactly how many times the loop should execute.

General form:

```
    while condition:
        statement(s)
```

Meaning: as long as the condition remains true, execute the statements.

As usual, all of the statements in the body must be indented the same amount.



While Loop

In file `WhileExample.py`:

```
COUNT = 500
STRING = "I will not throw paper airplanes in class."

def main():
    """ Print STRING COUNT times. """
    i = 0
    while ( i < COUNT ):
        print( STRING )
        i += 1

main()
```

```
> python WhileExample.py
I will not throw paper airplanes in class.
I will not throw paper airplanes in class.
  ...
I will not throw paper airplanes in class.
```

While Loop Example

Exercise: Find and print all of the positive integers less than or equal to 100 that are divisible by both 2 and 3.

While Loop Example

Exercise: Find and print all of the positive integers less than or equal to 100 that are divisible by both 2 and 3.

In file `DivisibleBy2and3.py`:

```
def main():
   num = 1
   while (num <= 100):
      if (num % 2 == 0 and num % 3 == 0):
         print( num, end=" " )
      num += 1
   print()

main()
```

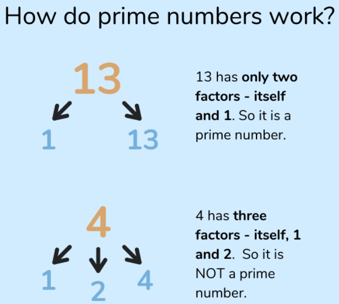
Running the program:

```
> python DivisibleBy2and3.py
6 12 18 24 30 36 42 48 54 60 66 72 78 84 90 96
>
```

Another While Loop Example: Test Primality

An integer is prime if it has no positive integer divisors except 1 and itself.

To test whether an arbitrary integer `n` is prime, see if any number in `[2 ... n-1]`, divides it.


How do prime numbers work?
13
1
13
13 has only two factors - itself and 1. So it is a prime number.
4
1
2
4
4 has three factors - itself, 1 and 2. So it is NOT a prime number.

You couldn't do that in *straight line* code without knowing `n` in advance. Why not?

Even then it would be *really* tedious if `n` is very large.

isPrime Loop Example

In file `IsPrime.py`:

```
def main():
   """ See if an integer entered is prime. """
   # Can you spot the inefficiencies in this?
   num = int( input("Enter an integer: ") )

   if ( num < 2 ):
      print (num, "is not prime")
   elif ( num == 2 ):
      print ("2 is prime")
   else:
      divisor = 2
      while ( divisor < num ):
         # Keep repeating this block until condition becomes
         # False, or exit if we find num is not prime.
         if ( num % divisor == 0 ):
            print( num, "is not prime" )
            return              # exit the function
         else:
            divisor += 1
      print(num, "is prime" )
```

isPrime Loop Example

```
> python IsPrime.py
Enter an integer: 53
53 is  prime
> python IsPrime.py
Enter an integer: 54
54 is not  prime
```

It works, though it's pretty inefficient. If a number is prime, we test every possible divisor in [2 ... n-1].

- We don't actually need the special test for 2. *Think about why that is.*
- If n is *not* prime, it will have a divisor less than or equal to $\sqrt{n}$.
- There's no need to test any even divisor except 2.

Exercise: Try for yourself writing a better version of this function.

For Loop

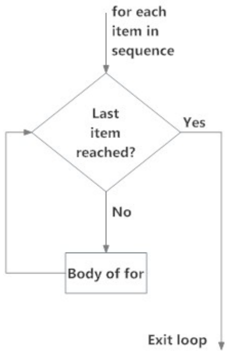
In a for loop, you typically know how many times you'll execute.

General form:

```
for var in sequence:
    statement(s)
```

Meaning: assign each element of sequence in turn to var and execute the statements.

As usual, all of the statements in the body must be indented the same amount.



What's a Sequence?

A Python sequence holds multiple items stored one after another.

```
seq = [2, 3, 5, 7, 11, 13]  # a list

sum = 0
for item in seq:
    sum += item

print( "The sum of the sequence is:", sum )
```

Range Function

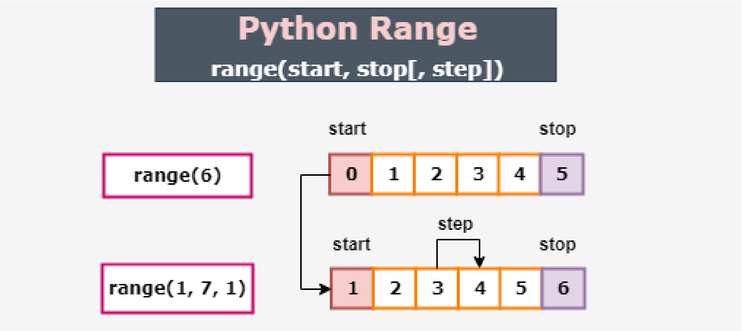
The range function is a good way to generate a sequence.

range(a, b) : denotes the sequence a, a+1, ..., b-1.

range(b) : is the same as range(0, b).

range(a, b, c) : generates a, a+c, a+2c, ..., b', where b' is the last value $<$ b.

Range Examples



Expression	Result
`range(3, 6)`	`3, 4, 5`
`range(3)`	`0, 1, 2`
`range(0, 11, 3)`	`0, 3, 6, 9`
`range(11, 0, -3)`	`11, 8, 5, 2`

Exercise: Explain why each of these turned out as it did.

For Example with Range

Exercise: Find and print all of the positive integers less than or equal to 100 that are divisible by both 2 and 3, using a `for` loop.

For Example with Range

Exercise: Find and print all of the positive integers less than or equal to 100 that are divisible by both 2 and 3, using a `for` loop.

```
def main():
    """ Print integers in [1..100] divisible by both 2 and 3.
        """
    for num in range(1, 101):
        if (num % 2 == 0 and num % 3 == 0):
            print( num, end=" " )
    print()
```

1. Why were the range limits 1 and 101?
2. What does the `end=" "` do? Would `end=""` work as well?
3. Why was the final `print()` there?

```
> python DivisibleBy2And3For.py
6 12 18 24 30 36 42 48 54 60 66 72 78 84 90 96
```

Another For Loop Example

Suppose you want to print a table of the powers of 2 up to 2^n.

In file `PowersOf2.py`:

```
def main():
    """ Print a table of powers of 2 up to 2**n,
        where n is entered by the user.  """
    num = int( input("Enter an integer: ") )

    for power in range (num + 1):      # Why num + 1
        print( power, "\t", 2 ** power )
```

Why does the range go to `num + 1`?

Another For Loop Example

```
> python PowersOf2.py
Enter an integer: 16
0        1
1        2
2        4
3        8
4        16
5        32
6        64
7        128
8        256
9        512
10       1024
11       2048
12       4096
13       8192
14       16384
15       32768
16       65536
```

Break and Continue

Two useful commands in loops (`while` or `for`) are:

break: exit the loop;

continue: exit the current iteration, but continue with the loop.

```
""" Square user inputs until a 0 is entered. """
while (True):
   num = int( input( "Enter an integer or 0 to stop: " ))
   if num == 0:
       break
   else:
       print( num ** 2 )
```

```
""" Print all numbers < 100 that are not multiples of 5. """
for num in range( 100 ):
    if num % 5 == 0:
       continue
    else:
       print( num )
```

Nested Loops

The body of `while` loops and `for` loops contain arbitrary statements, including other loops.

Suppose we want to compute and print out a multiplication table like the following:

```
          Multiplication Table
     |   1   2   3   4   5   6   7   8   9
------------------------------------------
   1 |   1   2   3   4   5   6   7   8   9
   2 |   2   4   6   8  10  12  14  16  18
   3 |   3   6   9  12  15  18  21  24  27
   4 |   4   8  12  16  20  24  28  32  36
   5 |   5  10  15  20  25  30  35  40  45
   6 |   6  12  18  24  30  36  42  48  54
   7 |   7  14  21  28  35  42  49  56  63
   8 |   8  16  24  32  40  48  56  64  72
   9 |   9  18  27  36  45  54  63  72  81
```

Multiplication Table

```
          Multiplication Table
     |   1   2   3   4   5   6   7   8   9
------------------------------------------
   1 |   1   2   3   4   5   6   7   8   9
   2 |   2   4   6   8  10  12  14  16  18
                  ....
   9 |   9  18  27  36  45  54  63  72  81
```

Here's an algorithm to do this:

1. How many columns/rows in the table?
2. Print the header information.
3. For each row i:
    1. Print i.
    2. For each column j: compute and print (i * j).
    3. Go to the next row.

This is easily coded using nested `for` loops.

Nested Loops

Print the header:

```
          Multiplication Table
    |   1   2   3   4   5   6   7   8   9
------------------------------------------
```

In file MultiplicationTable.py:

```
# Defines the size of the table + 1.
LIMIT = 10

def main():
    """ Print a multiplication table to LIMIT - 1. """
    print("          Multiplication Table")
    # Display the column headers.
    print("    |", end = "")
    for j in range(1, LIMIT):
        print(format(j, "4d"), end = "")
    print()    # jump to a new line
    # Print line to separate header from body of the table.
    print("------------------------------------------")
```

Nested Loops

This continues our multiplication example.

```
  1 |   1   2   3   4   5   6   7   8   9
  2 |   2   4   6   8  10  12  14  16  18
             ....
  9 |   9  18  27  36  45  54  63  72  81
```

```
    # Display table body
    for i in range(1, LIMIT):
        print( format(i, "3d"), "|", end = "")
        for j in range(1, LIMIT):
            # Display the product and align properly
            print( format( i*j, "4d"), end = "")
        print()

main()
```

Nested Loops Example

```
> python MultiplicationTable.py
          Multiplication Table
    |   1   2   3   4   5   6   7   8   9
------------------------------------------
  1 |   1   2   3   4   5   6   7   8   9
  2 |   2   4   6   8  10  12  14  16  18
  3 |   3   6   9  12  15  18  21  24  27
  4 |   4   8  12  16  20  24  28  32  36
  5 |   5  10  15  20  25  30  35  40  45
  6 |   6  12  18  24  30  36  42  48  54
  7 |   7  14  21  28  35  42  49  56  63
  8 |   8  16  24  32  40  48  56  64  72
  9 |   9  18  27  36  45  54  63  72  81
```

Notice that if you want a bigger or smaller table, you only have to change LIMIT in the code. But what would be wrong?