3.3 Loop Control Elements

Motivation

Even with all the different loop elements we've learned, occasionally loops are difficult to design so that they properly execute what we would like. Two statements that will help us with tricky loop situations are break and continue.

Break

Occasionally, we might be in a loop but run into a situation where we immediately want to exit the loop, instead of waiting until the loop runs its course. This is what the break statement allows us to do. Whenever we are inside of a loop and we hit a break statement, that immediately transfers control to the point right after the end of the loop.

Obviously, since we don't ALWAYS want to be broken out of a loop, it's nearly always the case that a break statement appears inside of an if statement inside of a loop.

Alternate Loop Design Using Break

Some programmers don't like checking the loop condition in the beginning of the loop. Instead, they set up each of their loops as follows:

```
while True:
```

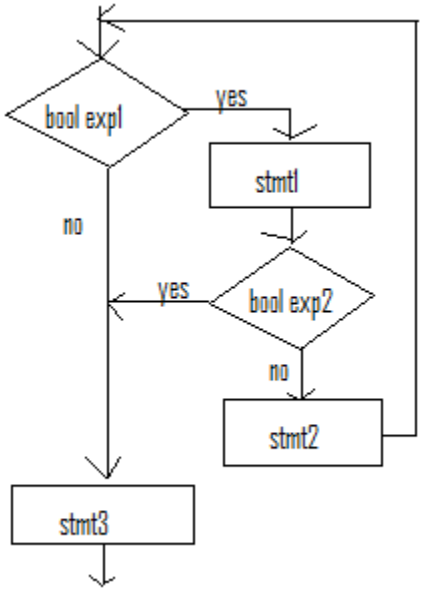
and use the break statement to get out of the loop whenever they want to. Similar to this:

```
while True:

    stmt1
    ...
    if <exit condition1>:
        break
    stmt2
    ...
    if <exit condition2>:
        break
    stmt3
    ...
```

Flow Chart Representation

As an example, consider the following segment of code and its flow chart representation:

```
while <bool expr1>:
    stmt1
    if <bool expr2>:
        break
    stmt2

stmt3
```



Prime Number Testing Example

A prime number is a number that is only divisible by 1 and itself. The standard method to test to see if a number is prime or not is to try dividing it by each number starting at 2, to see if there is any remainder or not. If any of these numbers divides evenly, meaning that it leaves no remainder, then the number is not prime.

In the following program, we will ask the user to enter a number and then determine whether it is prime or not. Once we find a number that divides evenly into the number our user has entered, there will be no further need to try other divisions.

```
def main():

    n = int(input("Please enter a number.\n"))

    # Try dividing this number by each possible divisor.
    isPrime = True
    for div in range(2, n):
        if n%div == 0:
            isPrime = False
            break

    # 2 is the smallest prime, so screen these out.
    if n < 2:
        isPrime = False

    # This number is prime, so print it.
    if isPrime:
        print(n,"is prime.")
    else:
        print(n,"is NOT prime.")

main()
```

The key idea here is that we try successively dividing n by 2, then 3, etc. If we ever find a number that divides evenly that is less than n, we immediately break. This means that instead of going back up to set div to the next value, we immediately skip to the next statement after the loop, the if statement. In essence, we stop the loop in its tracks when div equals the first value greater than one that divides evenly into n. If we never find such a value, the loop completes its course and isPrime is never set to false.

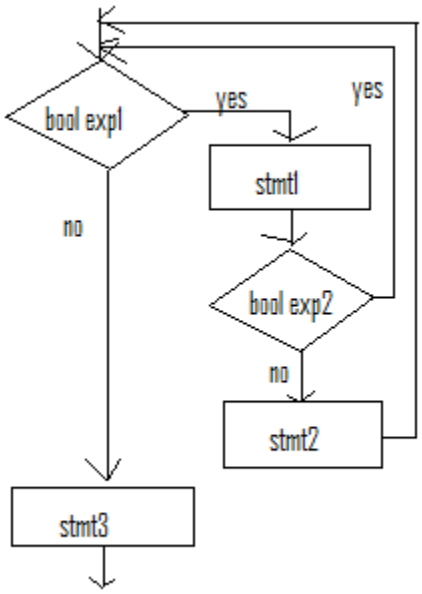
Continue

Continue is very similar to break except that instead of getting out of the loop completely, it simply skips over the rest of that following loop iteration and returns to the top of the loop. This may be desired if an input value can't be processed and a new input value is needed.

Flow Chart Representation

As an example, consider the following segment of code and its flow chart representation:

```
while <bool expr1>:
    stmt1
    if <bool expr2>:
        continue
    stmt2

stmt3
```



Notice that only one tiny change had to be made in this flow chart as compared to the flow chart illustrating the use of break.

Reading in Valid Test Scores Example

In the following example we will read in the first *n* valid test scores from the user and calculate their sum. Whenever the user enters an invalid score, we'll simply ignore it. We will use an if statement to screen out invalid test scores and continue in these situations.

Here is the program:

```
def main():

    total = 0

    n = int(input("How many valid test score to enter?\n"))

    print("Enter your scores, one per line.")
    print("Invalid scores will be ignored.")
    count = 0

    while count < n:

        score = int(input(""))
        if score < 0 or score > 100:
            continue

        count = count + 1
        total = total + score;

    print("Your valid scores add to ",total,".", sep="")

main()
```

Here, any time we read in an invalid score, we DON'T process it. Namely, we go back up to the top of the loop so that we can start the process to get the next score, without counting this one.