eCornell

**TOOL**

# Structure and Flow Concepts

This tool summarizes key concepts on conditionals, loops, and recursion that we have covered in this course. Use this document as a reference to refresh your understanding of program structure and program flow in Python.

## Using Conditional Statements

While program structure is the way statements are written in Python files, program flow is the order in which statements are executed. A conditional statement is one of the tools used to control program flow.

### If statement

**Execution:**

If the boolean expression is true, then execute all of the statements indented directly underneath (until first non-indented statement)

**Example:**

```
# Put x in z if it is positive
if x > 0:
   z = x
```

### If-else statement

**Execution:**

If the boolean expression is true, then execute statements indented under if; otherwise, execute the statements indented under else

**Example:**

```
# Put max of x, y in z
if x > y:
   z = x
else:
   z = y
```

### If-elif statement

**Execution:**

Python checks each boolean expression (starting with the if, and moving on to each elif) in order. For the first one that is true, it executes the statements indented under that line. If no boolean expression is true, it executes the statements indented under the else, if it exists.

**Example:**

```
# Put max of x, y, z in w
if x > y and x > z:
   w = x
elif y > z:
   w = y
else:
   w = z
```

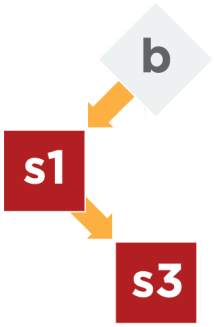
The else is always optional. If-elif by itself is fine.

## Conditional Branching

A conditional can almost be thought of as a flowchart. If an if statement is true, we execute statement s1 then go on to statement s3 outside the if statement. If it's false, we do not execute s1 and go straight to s3.

True

```
if b:
    s1 # statement
s3
```



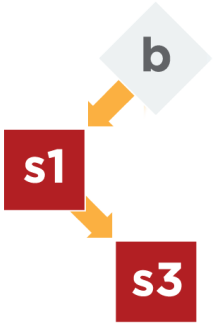

False

```
if b:
    s1 # statement
s3
```


If an if-else statement is true, we execute statement s1 then statement s3. If it's false, we execute s2 then s3.

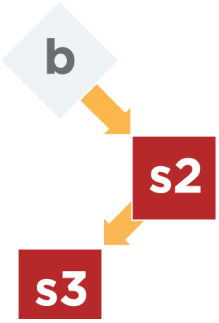
True

```
if b:
    s1 # statement
s3
```




False

```
if b:
    s1
else:
    s2
s3
```




Conditionals are a powerful programming tool when variables are unknown, such as when they are the result of user input and you don't know what a user is going to type or when the variables are the result of a function call. Your variables are parameters inside of a function and you don't know what arguments are going to be provided. Because of this, conditionals are a very natural fit for functions.

Sometimes the existence of a variable depends upon the flow of your code. This is why understanding program flow can be very important in testing your function.

# Using Try-Except Statements

## Try-except statement

The try-except is similar to the if-else statement. Python executes everything underneath the try. If nothing crashes, it skips over the except just like an else. But if any of the lines crash, it jumps to the except.

**Example:**

```
try:
   result = input('Number:') # get number
   x= float (input)          # convert to float
   print('The next number is'+str(x+1))
except:
   print('That is not a number!')
```

## Design Philosophy Difference

Conditionals are asking for permission

- Check if a property holds
- The body proceeds if it is safe

Try-except is asking for forgiveness

- Assumes that a property always holds
- Recovers if it does not

Python often prefers the latter

- This is largely unique to Python
- Errors are "relatively" cheap

## Print statements

Like conditionals, these can be used to trace statements to follow try-except.

**Example:**

```
def first(x):
   print('Starting first.')
   try:
      second(x)
   except:
      print('Caught at first')
   print('Ending first')

def second(x)
   print('Starting second.')
   try:
      third(x)
   except:
      print('Caught at second')
   print('Ending second')
```

## Testing functions with try-except

You have to look at the code and identify all of the possible ways that it can crash. Some may not be obvious. To have proper code coverage, look at everything that could cause a crash but still satisfies the precondition.

# Using For-Loops

## For-loop

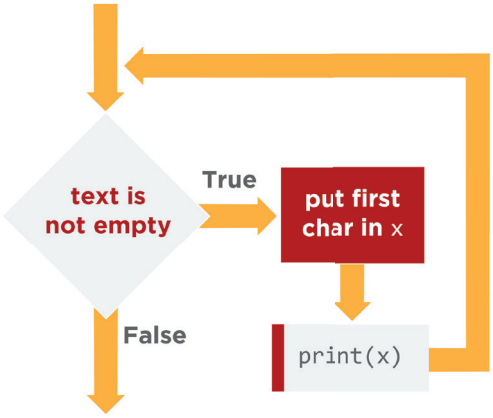
A for-loop is a control structure used for iterating over a sequence such as a tuple or a string.

The for-loop takes the first character of the iterable and puts it inside of the loop variable x. It then executes the body that is printing out x.

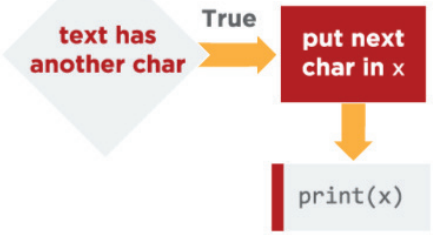
However, instead of moving on until later in the code, it goes all the way back to the for and now it takes the next character inside of the string and puts that inside of the variable x. It then repeats the body again. This process continues until there's nothing left in the string.

**Example:**

```
for x in text:
    print(x)
```




```
20      for x in text:
21          print(x)
22
```




## Tuple

- A sequence of arbitrary values between parentheses with all of the values separated by commas
- More common than strings
- Can be used in a for-loop
- Brackets are used to access its values

**Example:**

```
x = (5,6,5,9,15,23)

  x[0] is 5
  x[4] is 15
  x[0:2] is (5,6)
  x[3:] is (9,15,23)
```

**Two tricky things about tuples:**

- Cannot contain expressions, only values
- A tuple with just one element in it must have a comma after it, such as (4,). Otherwise, Python will evaluate it just as a number, not as a tuple.

Note: The IntroCS module has several powerful functions for working with tuples

## Accumulator

- Special variable that holds a final numeric answer
- The for-loop adds the numbers to it at each step
- Works on every type that can be added

## Range iterable

- Like a black box meant to be used inside of a for-loop to control the number of times the for-loop runs
- Defined by the range function
- Can be converted to a tuple

**Example:**

```
>>> range(3)
range(0,3)
>>> for x in range(3)
...     print(x)
0
1
2
>>> tuple(range(0,3))
(0, 1, 2)
```

# Using While-Loops

## While-loop

- Similar to a for-loop but able to run until it's finished
- For-loops have to run a fixed number of times

**Example:**

```
loop = True
while loop:
    try:
        result = input('Number: ')      # getnumber
        x = float(input)                # convert to float
        print('The next number is'+str(x+1))
        loop = False
    except:
        print('That is not a number! Try again')
```

Create loop variable

Update variable if successful

A problem with while-loops is that infinite loops are possible. For example, if you forget to update the loop variable or write the boolean expression incorrectly, the only possible action is to abort Python.

## Trace

- Print statement that shows where you are in your code
- Solution for detecting a while-loop error

**Example:**

```
print('Before while')
total = 0
x = 0
while x < n:
   print('Start loop'+str(x))
   total = total + x*x
   x = x + 1
   print('End loop')
print('After while')
```

**Output:**

```
Before while
Start loop 0
End loop
Start loop 1
End loop
Start loop 2
End loop
After while
```

While-loops are a very powerful tool and especially useful in applications that are interactive with a user, like games.

**Example:**

```
def roll_past(goal):
   ""Returns: The score from rolling a die until
   passing goal.""
   loop= True  #Keep looping until this is false
   score = 0
   while loop:
      roll = random.randint(1,6)
      if roll == 1:
         score = 0; loop = False
      else:
         score = score + roll; loop = score < goal
   return score
```