DSC 510

Week 3 Conditional execution And Exception Handling

Getting User Input

This is the typical flow of a simple computer program:

1. Input data.
2. Work with data.
3. Do some computation(s).

Output some answer(s).In Python, we can get input from the user using a built-in function called raw_input. Here's how it is used, most typically in an assignment statement:

```
<variable> = input(<prompt string>)
```

On the right side of the equals sign is the call to the raw_input built-in function. When you make the call, you must pass in a *prompt string*, which is any string that you want the user to see. The prompt is a question that you want the user to answer. The raw_input function returns all of the characters that the user types, as a string. Here is an example of how raw_input might be used in a program:

```
favoriteColor = input('What is your favorite color? ')
Print('Your favorite color is', favoriteColor )
```

Assignment Statement

When the assignment statement runs, the following steps happen in order:

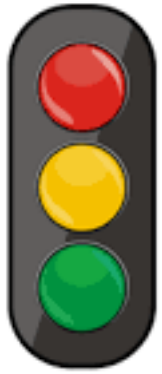
1. The prompt string is printed to the Shell.
2. The program stops and waits for the user to type a response.
3. The user enters some sequence of characters into the Shell as an answer.
4. When the user presses the Enter key (Windows) or the Return key (Mac), input() returns the characters that the user typed.
5. Typically, input() is used on the right side of an assignment statement. The user's response is stored into the variable on the left-hand side of the equals sign.

Conditional Statements

- Programming often involves examining a set of conditions and deciding which action to take based on those conditions. Python's `if statement` allows you to examine the current state of a program and respond appropriately to that state.

If Statements are Everywhere!

The Traffic Light: An Everyday IF Statement



IF light is **red**, THEN stop!

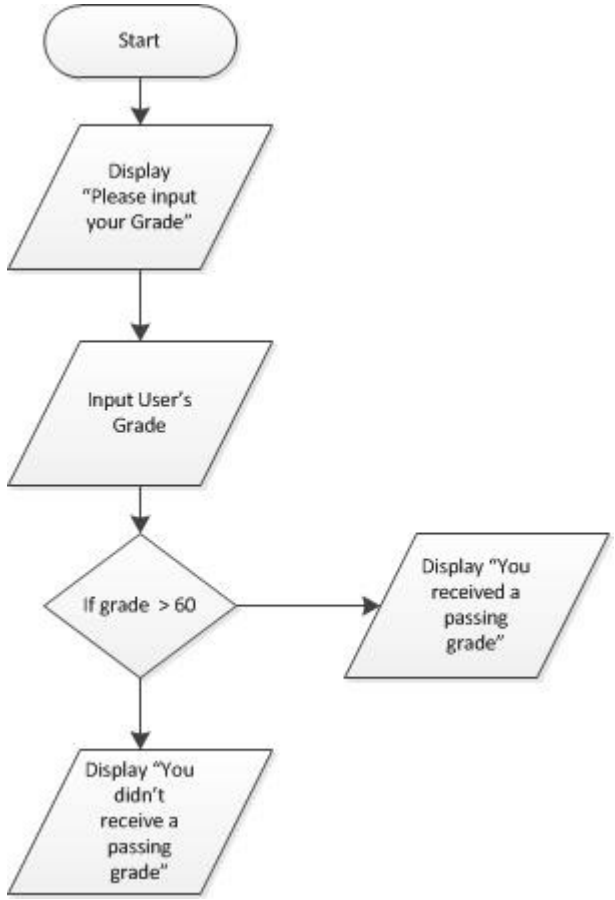
IF light is **yellow**, THEN what???

IF light is **green**, THEN go!

Example

Flowcharts are an effective way to visually show the program flow when if statements are used. In the flowchart example on the left we have an `if statement` represented by the diamond shape.

In this example we evaluate whether or not the grade input by the user is over 60. If the result of the comparison is true we display “You received a passing grade”, if it is false, we display “You didn’t receive a passing grade”.



If Statement Syntax

```
if <Boolean expression>: # notice the colon at the end of the line
    <indented block of code> # any number of indented lines
```

Example:

```
if grade > 60:
    print("You received a passing grade")
```

Else Statement

It is likely that we will want the program to do something even when an if statement evaluates to false. In our grade example, we will want output whether the grade is passing or failing. It is incredibly common to have an else follow an if statement so that a standard behavior is triggered if the if condition is not met.

To do this, we will add an else statement to the grade condition above that is constructed like this:

```
grade = input("Please input your Grade: ")

if grade > 60:
    print("You received a passing grade")
else:
    print("You didn't receive a passing grade")
```

Else If Statement

In many cases, we will want a program that evaluates more than two possible outcomes. For this, we will use an **else if** statement, which is written in Python as `elif`. The `elif` or else if statement looks like the if statement and will evaluate another condition.

```
balance = input(“Please enter your balance”)

if balance < 0:
    print("Balance is below 0, add funds now or you will be charged a penalty.")
elif balance == 0:
    print("Balance is equal to 0, add funds soon.")
else:
    print("Your balance is 0 or above.")
```

Comparison Operators

There are multiple comparison operators which can be used in Python `if statements`:

Operator	Meaning	Example
==	equals	if a == b:
!=	not equals	if a != b:
<	less than	if a < b:
>	greater than	if a > b:
<=	less than or equal to	if a <= b:
>=	great than or equal to	if a >= b:

Nested If Statement

When a Boolean expression in an if statement evaluates to True, then the indented block of code runs. But the indented block of code can contain another if statement. And if the Boolean expression in an indented if statement also evaluates to True, then the indented code block associated with that if statement will run; for example:

```
# Purchases at a gas station
totalGasPurchase = priceOfGas * nGallons
amountLeftOver = startingAmountOfMoney – totalGasPurchase

# See if we have enough money to buy a Powerball lottery ticket
if amountLeftOver > 2:
    feeling = evaluateEmotions()
    if feeling == 'lucky':
     buyPowerballTicket()
     amountLeftOver = amountLeftOver - 2
```

Example Program Using Nested If Statements

```
priceOfGas = input('Please enter the price of gas: ')
nGallons = input('Please enter the number of gallons purchased: ')
startingAmountOfMoney = input('How much money do you have in your pocket: ')
# Purchases at a gas station
totalGasPurchase = float(priceOfGas) * float(nGallons)
amountLeftOver = float(startingAmountOfMoney) - totalGasPurchase
# See if we have enough money to buy a Powerball lottery ticket
print ('The total gas purchase is ', round(totalGasPurchase, 2))
print ('Your change is ', round(amountLeftOver, 2))
if (amountLeftOver >= 25):
   print ('We have a bunch of money left over!!')
   dinner = input('Should we go out to dinner? Enter Y for Yes and N for No: ')
   print(dinner)
   if (dinner == 'Y' or dinner == 'y'):
      restaurantChoice = input('Where do you want to eat? ')
      print ('Heading to: ', restaurantChoice)
   else:
      print ('Heading home.')
else:
   print ('Looks like we should save some money and go home!')
```

Indentation

In many programming languages indentation doesn't matter because a block of code is typically encapsulated within brackets {}. In Python however, it is important that blocks of code be indented the same number of spaces. If a particular line of code within a block doesn't have the appropriate spacing the following error will be given. In this case you'll need to check the indentation of the lines of code surrounding the error:

```
    C:\CIS245>python nestedif.py
 File "nestedif.py", line 14
    if (dinner == 'Y' or dinner =='y'):
                                      ^
TabError: inconsistent use of tabs and spaces in indentation
```

To resolve issues such as above you must check that the indentation within the If statement block of code is consistent. Many modern IDEs have tools to help with locating and resolving these issues.

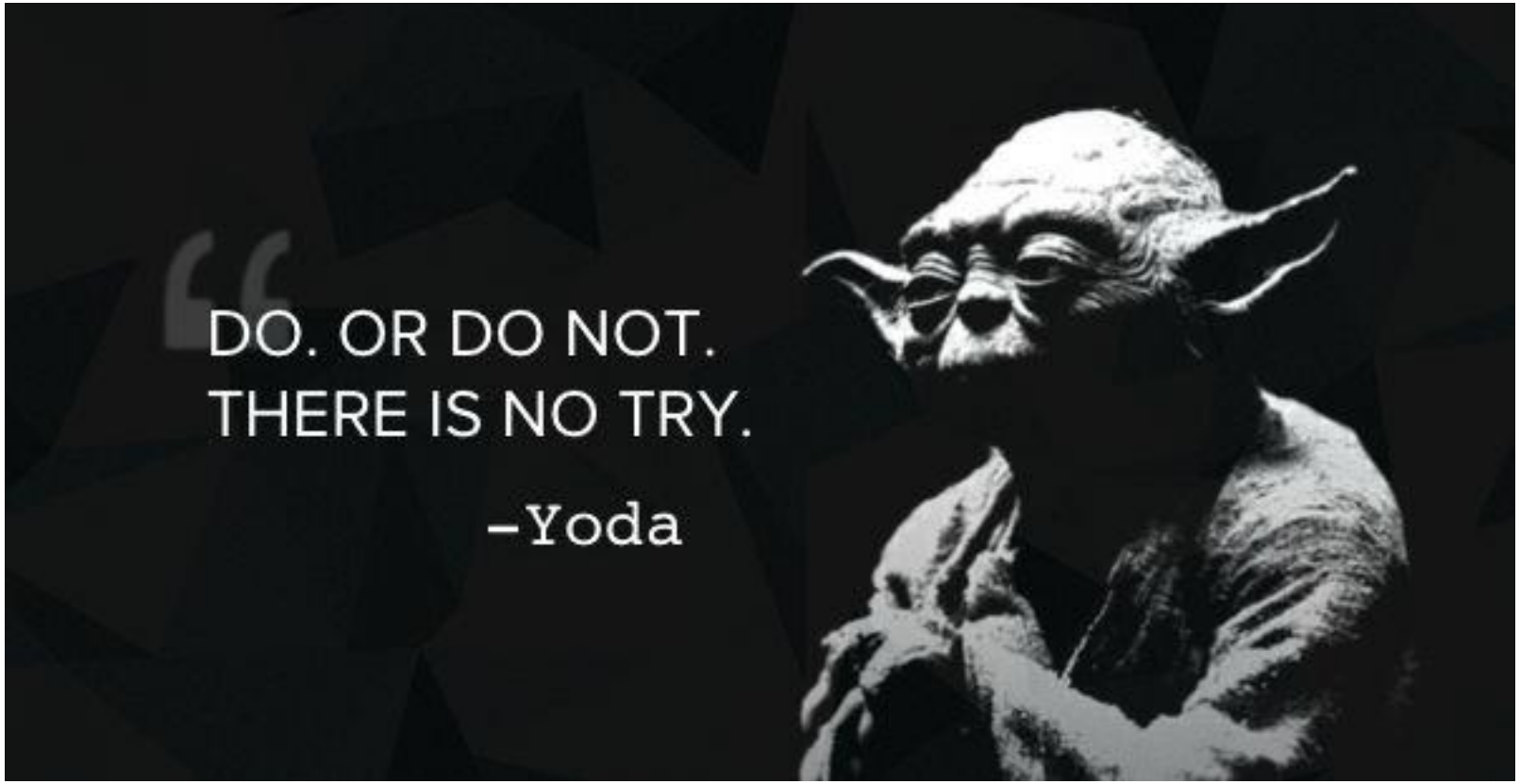
Python Exception Handling

In programing we typically encounter two types of issues with our code; syntax errors and exceptions. Syntax errors prevent the execution of the program until they're corrected and occur when the rules of the Python language are not followed. An example of a syntax error would be excluding the ":" after the first line of an **If Statement** such as:

```
if (amountLeftOver >= 25) #notice we're missing the :
```

Exceptions are created during the execution of a program and although don't necessarily stop the program from running can cause unexpected behavior. The good thing about most programming languages including Python is that we have the ability to handle exceptions without our programs.

Except in Programming…



Try Except

In Python we have the ability to use a Try Except block to catch exceptions in our programs. The try statement works as follows:

- *First, the try clause (the statement(s) between the try and except keywords) is executed.*
- *If no exception occurs, the except clause is skipped and execution of the try statement is finished.*
- *If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.*
- *If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above.*
- *A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement.*

Try Except Syntax

```
try:
    <statement(s) that may cause an error>
except:
    <statement(s) to execute IF an error occurs>
else:    #optional, often not needed
    <statement(s) to execute if NO error occurs>
finally:
   <statement(s) to execute in all cases>
```

Try/Except Example

```
# Check for potential error
# if the userInput isn’t an Integer, catch the exception and print a message

userInput = input('Please enter an integer: ')
try:
    userInput = int(userInput)
except ValueError:
    print 'The number you entered was not an integer'
    # Code here to alter execution because we do not want to keep going
```

BaseException Class

The BaseException class is the base class of all the exceptions. It has four sub-classes.

- **Exception** – this is the base class for all non-exit exceptions.
- **GeneratorExit** – Request that a generator exit.
- **KeyboardInterrupt** – Program interrupted by the user.
- **SystemExit** – Request to exit from the interpreter.

- The Class Hierarchy of Python Exceptions can be found here:
  - https://docs.python.org/3/library/exceptions.html#exception-hierarchy

Built-in exception classes in Python

- **ArithmeticError** – this is the base class for arithmetic errors.
- **AssertionError** – raised when an assertion fails.
- **AttributeError** – when the attribute is not found.
- **BufferError**
- **EOFError** – reading after end of file
- **ImportError** – when the imported module is not found.
- **LookupError** – base exception for lookup errors.
- **MemoryError** – when out of memory occurs
- **NameError** – when a name is not found globally.
- **OSError** – base class for I/O errors
- **ReferenceError**
- **RuntimeError**
- **StopIteration**, **StopAsyncIteration**
- **SyntaxError** – invalid syntax
- **SystemError** – internal error in the Python Interpreter.
- **TypeError** – invalid argument type
- **ValueError** – invalid argument value

Try/Except Best Practices

- Catch Specific Exceptions
- Keep try blocks small
  - Don’t cram huge blocks of code into single try blocks
- Use Else and Finally clauses when necessary

Cyclomatic Complexity

Cyclomatic complexity is a software metric used to measure the complexity of a program. These metric, measures independent paths through program source code.

Independent path is defined as a path that has at least one edge which has not been traversed before in any other paths.

Cyclomatic complexity can be calculated with respect to functions, modules, methods or classes within a program.

This metric was developed by Thomas J. McCabe in 1976 and it is based on a control flow representation of the program. Control flow depicts a program as a graph which consists of Nodes and Edges.

In programming things such as multiple imbedded if statements can lead to Cyclomatic complexity. An increased Cyclomatic complexity leads to degraded efficiency of the application as well as making the program more difficult to read. When designing a program especially if statements we need to ensure that our program isn't becoming too complex.