Choice

- Depending on the outcome of a condition, you execute one set of instructions or another
- This is how computer programs appear to have “intelligence”
The if Statement

Syntax:

```python
if(condition):
    code block
elif (condition):
    code block
else:
    code block
```

An Example

• To Compute Gross Pay:
  – if hours worked is more than 40, sum the hourly rate multiplied by 40 with the number of hours worked over 40 multiplied by 1.5 times the hourly rate
  – If hours worked is 40 or less, multiply the number of hours worked by the hourly rate
• Call the result gPay
Computing Gross Pay

```python
hours = int(input("hours? "))
rate = float(input("rate? "))
if(hours > 40):
    gPay = 40 * rate
    gPay = gPay + (hours-40) * (rate*1.5)
else:
    gPay = hours * rate
print("Gross Pay: ",gPay)
```

A More Complicated Example to Compute Gross Pay

- if hours worked is more than 80, sum the hourly rate multiplied by 40, with the hourly rate times 1.5 multiplied by 40, and the hours worked over 80 multiplied by the hourly rate times * 2
- if hours worked is more than 40 but less than 80, sum the hourly rate multiplied by 40 with the number of hours worked over 40 multiplied by 1.5 times the hourly rate
- If hours worked is 40 or less, multiply the number of hours worked by the hourly rate
- Call the result gPay
Computing a More Complicated Gross Pay

hours = int(input("hours? "))
rate = float(input("rate? "))
if(hours > 80):
gPay = 40 * rate
gPay = gPay + 40 * (rate*1.5)
gPay = gPay + (hours-80) * (rate*2.0)
elif(hours > 40):
gPay = 40 * rate
gPay = gPay + (hours-40) * (rate*1.5)
else:
gPay = hours * rate
print("Gross Pay: ",gPay)

My First Try

hours = int(input("hours? "))
rate = float(input("rate? "))
if(hours > 80):
gPay = 40 * rate
gPay = gPay + 40 * (rate*1.5)
gPay = gPay + (hours-80) * (rate*1.5)
elif(hours > 40):
gPay = 40 * rate
gPay = gPay + (hours-40) * (rate*1.5)
else:
gPay = hours * rate
print("Gross Pay: ",gPay)
Code Inspection: *I thought I made a mistake once, but I was wrong*

- Always a good idea to look your code over for obvious mistakes
- If the main part of the code is too large to fit on a single screen, *print it out to paper*
- Peer inspections are commonly used
  - sit down and explain your program’s execution to someone else – they probably won’t find an error, but in the process of explaining it, you probably will

Testing to See If Your Program Works Correctly

- Testing is feeding your program some pre-determined inputs, *for which you have already calculated the expected results* to see if you get what you expected
- Possible Results:
  - **Correct** – actual results match your expected results
  - **Run-time Error** – program “crashes”
  - **Logic Error** – actual results do not match your expected results
Common Approach to Testing

case-based testing

• What are the various cases your program expects?
  – Worked 40 hours or less
  – Worked more than 40 hours but less than 80 hours
  – Worked more than 80 hours

• Your test cases:
  – hours = 30, rate = 12, expect: 360
  – hours = 45, rate = 12, expect: 570
  – hours = 90, rate = 12, expect: 1440

Specification Error

• What if the hours worked is 80 exactly?
  – if hours worked is more than 80 … if hours worked is more than 40 but less than 80 …

• Very common for the problem statement to be incorrect – if you were to implement this specification, your program would be technically correct, but the user would still not be happy

• Peer inspections work with specifications too
Logical Data Types

• Conditions evaluate to True or False:
  
  \[ \text{if}(\text{hours} > 80): \]

• You can have a variable that points to a location containing a True or False value

  \[ \text{forever} = \text{True} \]
  
  \[ \text{while}(\text{forever}): \]
  
  \[ \text{print(“animal crackers”)} \]

Negation

• You can use `not` to “negate” a logical value:

  \[ \text{forever} = \text{True} \]
  
  \[ \text{while}(\text{not} \ \text{forever}): \]
  
  \[ \text{print(“animal crackers”)} \]

• If `x` is True, `\text{not} \ x` is False
• if `x` is False, `\text{not} \ x` is True
Imply Logical Values from Numeric/String Values

- Zero is false
- The empty string “” is false
- Everything else is True

*Use sparingly – you should use explicit logical comparisons until you really understand what you are doing*

Compound Conditions

- Simple conditions usually consist of two values connected by a relational operator:

  ```python
  if(hours > 80):
  ```

- What if multiple conditions must be met?
- If the “employee works over 80 hours and belongs to the widget transporter union they get double time for every hour worked over 80”
Logical Operators

- and
- or
- logical operators connect two relational expressions

Check for hours over 80 AND a member of the union

```python
unionQ = input("member of the union(Y/N)? ")
if(unionQ == "Y"):
    union=True
else:
    union=False
hours = int(input("hours? "))
rate = float(input("rate? "))
if((hours > 80)and(union)):
    gPay = 40 * rate
    gPay = gPay + 40 * (rate*1.5)
    gPay = gPay + (hours-80) * (rate*2.0)
else:
    gPay = hours * rate
print("Gross Pay: ",gPay)
```
Check for hours over 80 AND a member of the union - alternate

```python
unionQ = input("member of the union (Y/N)? ")
hours = int(input("hours? "))
rate = float(input("rate? "))
if((hours > 80) and (unionQ == "Y")):
    gPay = 40 * rate
    gPay = gPay + 40 * (rate*1.5)
    gPay = gPay + (hours - 80) * (rate*2.0)
else:
    gPay = hours * rate
print("Gross Pay: ", gPay)
```

Check for union membership, or boss' family member

```python
union = input("member of union (Y/N)? ")
family = input("related to boss (Y/N)? ")
hours = int(input("hours? "))
rate = float(input("rate? "))
if((hours>80) and ((union=="Y") or (family=="Y"))):
    gPay = 40 * rate
    gPay = gPay + 40 * (rate*1.5)
    gPay = gPay + (hours-80) * (rate*2.0)
else:
    gPay = hours * rate
print("Gross Pay: ", gPay)
```
Syntax of a logical expression

- `{relational expr} {logical op} {relational expr}`
- `{logical op}` is and or or
- Any place you can have a relational expression, you can have a logical expression
- if, while, etc.
- Logical expressions have an order to evaluation: left to right with and before or
- Use parentheses for grouping expressions – make sure they are balanced

Truth Tables

<table>
<thead>
<tr>
<th>Relop1</th>
<th>Relop2</th>
<th>Relop1 and Relop2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relop1</th>
<th>Relop2</th>
<th>Relop1 or Relop2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Truth Tables with *Negation*

<table>
<thead>
<tr>
<th>Relop1</th>
<th>Relop2</th>
<th>not(Relop1 and Relop2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relop1</th>
<th>Relop2</th>
<th>not(Relop1 or Relop2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

Strings and Numerics

- `input()` always returns a string
- If we want to do arithmetic, we need to do a type conversion to numeric using `int()`:
  
  ```python
  aNumber = int(input("enter number"))
  ```
- Works great if I enter “1234” but what if I type in “apple pie?”
Trying to Convert “apple pie” to a number – won’t work

```python
>>> enter number apple pie
Traceback (most recent call last):
  File "C:/Users/Warren/Dropbox/Courses/CS161/Python Code/stringNints.py", line 1, in <module>
    aNumber=int(input("enter number "))
ValueError: invalid literal for int() with base 10: 'apple pie'
```
isnumeric()

• a method() that can be applied to strings
• We use a method to apply an operation to an object such as a string.
• isnumeric() returns a True or False
• Syntax: `stringVar.isnumeric()`
• Semantics: return TRUE if `stringVar` holds a value that can be converted into a number, otherwise return FALSE

Example

```python
stringVar=input("enter number ")
if stringVar.isnumeric():
    print("you entered a number")
else:
    print("you did not enter a number")
    stringVar=input("enter number ")
print("the number is ",stringVar)
```
Revised Example Using Negation

```python
stringVar=input("enter number ")
if not(stringVar.isnumeric()):
    print("you did not enter a number")
    stringVar=input("enter number ")
print("the number is ",stringVar)
```