Clarifying PEMDAS
A Few Variations on Things We’ve Seen
Types We **Have** Seen

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Numbers</td>
<td>x = 32</td>
</tr>
<tr>
<td>Floating Point Numbers</td>
<td>x = 212.00</td>
</tr>
<tr>
<td>Strings</td>
<td>x = ‘forgetMeNot’</td>
</tr>
</tbody>
</table>
Types We Haven’t Seen

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean Values</td>
<td>bool</td>
</tr>
<tr>
<td></td>
<td>x = True</td>
</tr>
<tr>
<td></td>
<td>y = False</td>
</tr>
<tr>
<td>List of Values</td>
<td>list</td>
</tr>
<tr>
<td></td>
<td>x = [4, 3, 1, 2]</td>
</tr>
<tr>
<td>Dictionary</td>
<td>dict</td>
</tr>
<tr>
<td></td>
<td>y = {'first': 'Yigal', 'last': 'Arens'}</td>
</tr>
</tbody>
</table>

\[x[2] \text{ is } 1\]
\[y[\text{‘last’}] \text{ is } \text{‘Arens’}\]
More Printing

• What happens if we need to print these lines?
  I am 5’9” tall
  I want to see the 5’9\” backslash!
# Characters We Haven’t Seen

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>What It Does</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\</code></td>
<td>Backslash ()</td>
</tr>
<tr>
<td><code>'</code></td>
<td>Single-quote (')</td>
</tr>
<tr>
<td><code>&quot;</code></td>
<td>Double-quote (&quot;)</td>
</tr>
<tr>
<td><code>\a</code></td>
<td>ASCII bell (BEL)</td>
</tr>
<tr>
<td><code>\b</code></td>
<td>ASCII backspace (BS)</td>
</tr>
<tr>
<td><code>\n</code></td>
<td>ASCII linefeed (LF)</td>
</tr>
<tr>
<td><code>\r</code></td>
<td>Carriage Return (CR)</td>
</tr>
<tr>
<td><code>\t</code></td>
<td>Horizontal Tab (TAB)</td>
</tr>
</tbody>
</table>
Conditional Execution, Functions
Conditional Steps

Program:

```python
x = 5
if x < 10:
    print('Smaller')
if x > 20:
    print('Bigger')
print('Finis')
```

Output:

Smaller
Finis
Conditional Execution

- So far we’ve seen Python execute lines in order
  - One line of code after the next, whether in the interpreter or from a file
    ```python
    >>> x = "10"
    >>> y = x + "1"
    >>> print(y)
    ```
  - Conditional execution allows you to say
    - “Only if this is true, do that”
  - But how do we know if something is true?
  - Let’s spend some time on that question
Boolean Variables

- Boolean (bool) is another type of variable (like int, float, str) but it can only take two values:
  - True or False (note capitalization!)

    >>> type(True)
    <type 'bool'>

- By the way, what are the outputs of the following?

    >>> type("True")
    >>> type(true)
Comparison Operators

- *Comparison operators* operate on *operands* which are Python variables or constants. The comparison operators are:

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x == y</code></td>
<td>is equal to</td>
</tr>
<tr>
<td><code>x != y</code></td>
<td>not equal to</td>
</tr>
<tr>
<td><code>x &gt; y</code></td>
<td>greater than</td>
</tr>
<tr>
<td><code>x &lt; y</code></td>
<td>smaller than</td>
</tr>
<tr>
<td><code>x &gt;= y</code></td>
<td>greater than or equal to</td>
</tr>
<tr>
<td><code>x &lt;= y</code></td>
<td>less than or equal to</td>
</tr>
<tr>
<td><code>x is y</code></td>
<td>is the same as</td>
</tr>
<tr>
<td><code>x is not y</code></td>
<td>is not the same as</td>
</tr>
</tbody>
</table>

- Each of the above is a “Boolean expression”
- Boolean expressions ask a question, to which the answer is “yes” or “no”, i. e., True or False. They do not change values.
Equality Is Not Assignment!

• These are not the same:
  
  my_var = 10
  my_var == 10

• If you confuse the two, Python will either let you know (e. g., it’s syntax error), or your program will fail in another way…
Logical Operators

- *Logical operators* operate on Boolean expressions/values
  - On things like \((x == \text{‘Bobby’}), \text{False}, (y > 2)\)
- There are three logical operators:
  - `and`
  - `or`
  - `not`
- Their meaning in Python is essentially their common-sense meaning
- Are you familiar with “truth tables”? 

Logical Operator Caveats

- You can plug just about anything into a logical operator
  - For example, nonzero integers will evaluate to True.
- You do not want to do this!
- Stick with things you know evaluate explicitly to Boolean variables (i.e. Boolean expressions)
Conditional Execution

```python
if x == 5:
    print ('x is definitely 5')
print ('x may or may not be 5')
```

- Note the colon!
- Note the indentation!
- Why does the last line above say what it does?
Indentation

- Increase indent indent after an if statement or for statement (after : )
- Maintain indent to indicate the scope of the block (which lines are affected by the if/for)
- Reduce indent back to the level of the if statement or for statement to indicate the end of the block
- Blank lines are ignored - they do not affect indentation
- Comments on a line by themselves are ignored with regard to indentation
Warning: Turn Off Tabs!!

• Atom automatically uses spaces for files with ".py" extension (nice!)

• Most text editors can turn tabs into spaces - make sure to enable this feature
  
  - NotePad++: Settings -> Preferences -> Language Menu/Tab Settings
  - TextWrangler: TextWrangler -> Preferences -> Editor Defaults

• Python cares a *lot* about how far a line is indented. If you mix tabs and spaces, you may get “indentation errors” even if everything looks fine
This will save you much unnecessary pain.
increase / maintain after if or for
decrease to indicate end of block

```python
x = 5
if x > 2 :
    print('Bigger than 2')
    print('Still bigger')
print('Done with 2')

for i in range(5) :
    print(i)
    if i > 2 :
        print('Bigger than 2')
    print('Done with i', i)
print('All Done')
```
x = 5
if x > 2 :
    print('Bigger than 2')
    print('Still bigger')
print('Done with 2')

for i in range(5) :
    print(i)
    if i > 2 :
        print('Bigger than 2')
        print('Done with i', i)
    print('All Done')
Try Some Code
Else

• With an *else* statement, only one *block* will be executed, either the *if* block or the *else* block

```python
if age >= 18:
    print (‘you can vote!’)
else:
    print ("you’re too young to vote!")
```

• (Note the use of double quotes. Why did I do that?)

• Same indentation rules apply to *else* as to *if*

• And note the colon again!
Two-way Decisions

- Sometimes we want to do one thing if a logical expression is true and something else if the expression is false.
- It is like a fork in the road - we must choose one or the other path but not both.
Two-way Decisions with else:

```python
x = 4

if x > 2:
    print('Bigger')
else:
    print('Smaller')

print('All done')
```

![Flowchart diagram](image-url)
x = 4

if x > 2 :
    print('Bigger')
else :
    print('Smaller')

print('All done')
Chained Conditionals – *elif*

- *elif* can be thought of as “else if”
- If this is true <do something>, otherwise, if this is true <do something else> otherwise, if this is true <do something else> . . .
- Note: One of an *if/else* has to execute. That is not true of *elif*
  - But an *else* at the end will catch the case where all the ones above were false
if $x < 2$ :
    print('small')
elif $x < 10$ :
    print('Medium')
else :
    print('LARGE')
print('All done')

Multi-way
Multi-way

```python
x = 0
if x < 2:
    print('small')
elif x < 10:
    print('Medium')
else:
    print('LARGE')
print('All done')
```
`x = 5
if x < 2 :
    print('small')
elif x < 10 :
    print('Medium')
else :
    print('LARGE')
print('All done')`
x = 20
if x < 2 :
    print('small')
eelif x < 10 :
    print('Medium')
else :
    print('LARGE')
print('All done')

\[ x = 20 \]
\[ x < 2 \]
\[ \text{yes} \rightarrow \text{print('small')} \]
\[ \text{no} \]
\[ x < 10 \]
\[ \text{yes} \rightarrow \text{print('Medium')} \]
\[ \text{no} \]
\[ \text{print('LARGE')} \]
\[ \text{print('All Done')} \]
Can End with Else, or Not

Multi-way

# No Else
x = 5
if x < 2 :
    print('Small')
elif x < 10 :
    print('Medium')
print('All done')

if x < 2 :
    print('Small')
elif x < 10 :
    print('Medium')
elif x < 20 :
    print('Big')
elif x < 40 :
    print('Large')
elif x < 100:
    print('Huge')
else :
    print('Ginormous')

But note that ‘All done’ will always print!
try/except

- Sometimes things do not go according to plan
- For example, you may get input you don’t expect

# Convert Fahrenheit to Celsius
f_temp = input('Temp in Fahrenheit ')
f_temp = float(f_temp)
c_temp = (f_temp - 32.0) * 5.0 / 9.0
print (f_temp, 'Fahrenheit is', c_temp, 'Celsius')

- What if someone types something other than a number?
- Let’s check it out
try/except

try:
    <a block of code>
except:
    <a block of code that will be executed only if there was an error in the block above>

• try/except blocks are in some sense similar to if/else blocks

• A statement in the “try” block is said to throw the exception, and the “except” block catches it
Functions

- Often you want to repeat blocks of code in your program
  - For example, you may need to convert temperature in Fahrenheit to Celsius multiple times
- It would be wasteful to have to include the same code multiple times
- A function may take input(s) and may provide output
Examples of Functions

- We’ve already seen several *built-in* functions, e. g.,
  - `int('12')`, `input("> ")`, `type("string")`, `float(6)`
  - Do they take input(s)? Do they have output?

- There are many other built-in functions, for instance:
  - `max()`, `min()`
Python Functions

- There are two kinds of functions in Python.
  - **Built-in functions** that are provided as part of Python - `print()`, `input()`, `type()`, `float()`, `int()` ...
  - **Functions that we define ourselves** and then use

- We treat the built-in function names as “new” **reserved words** (i.e., we avoid them as variable names)
Function Definition

• In Python a function is some reusable code that takes arguments(s) as input, does some computation, and then returns a result or results

• We define a function using the def reserved word

• We call/invoke the function by using the function name, parentheses, and arguments in an expression
big = max('Hello world')

w

tiny = min('Hello world')

w

>>> big = max('Hello world')
>>> print(big)
w
>>> tiny = min('Hello world')
>>> print(tiny)
Max Function

```python
>>> big = max('Hello world')
>>> print(big)

'w'
```

A function is some stored code that we use. A function takes some input and produces an output.

Guido wrote this code
Max Function

A function is some stored code that we use. A function takes some input and produces an output.

Guido wrote this code

```python
def max(inp):
    blah
    for x in inp:
        blah

'Hello world' (a string) -> 'w' (a string)
```
Building our Own Functions

• We create a new function using the `def` keyword followed by optional parameters in parentheses

• We indent the body of the function

• This **defines** the function but **does not** execute the body of the function

```python
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.")
```
Arguments

• An argument is a value we pass into the function as its input when we call the function

• We use arguments so we can direct the function to do different kinds of work when we call it at different times

• We put the arguments in parentheses after the name of the function

   \[ \text{big} = \max(\text{'Hello world'}) \]
Parameters

A parameter is a variable which we use in the function definition. It is a “handle” that allows the code in the function to access the arguments for a particular function invocation.

```python
>>> def greet(lang):
...     if lang == 'es':
...         print('Hola')
...     elif lang == 'fr':
...         print('Bonjour')
...     else:
...         print('Hello')
...

>>> greet('en')
Hello
>>> greet('es')
Hola
>>> greet('fr')
Bonjour
```
Return Values

Often a function will take its arguments, do some computation, and return a value to be used as the value of the function call in the calling expression. The return keyword is used for this.

```python
def greet():
    return "Hello"

print(greet(), "Glenn")
print(greet(), "Sally")
```

Hello Glenn
Hello Sally
Return Value

- A “fruitful” function is one that produces a result (or return value)

- The return statement ends the function execution and “sends back” the result of the function

```python
>>> def greet(lang):
...     if lang == 'es':
...         return 'Hola'
...     elif lang == 'fr':
...         return 'Bonjour'
...     else:
...         return 'Hello'
...
>>> print(greet('en'),'Glenn')
Hello Glenn
>>> print(greet('es'),'Sally')
Hola Sally
>>> print(greet('fr'),'Michael')
Bonjour Michael
>>>
Multiple **Parameters** / **Arguments**

- We can define more than one parameter in the function definition.
- We simply add more arguments when we call the function.
- We match the number and order of arguments and parameters.

```python
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print(x)
```

8
Void (non-fruitful) Functions

• When a function does not return a value, we call it a "void" function

• Functions that return values are "fruitful" functions

• Void functions are "not fruitful"
Functions Must Be Defined Before Use

• Built-in functions are all predefined and may be used at any time

• Python has many modules that contain function definitions — you must *load* those modules before using their functions

  • See section 4.5 in the book for how to get many math functions

• You must define your own functions before you can use them
To function or not to function...

- Organize your code into “paragraphs” - capture a complete thought and “name it”
- Don’t repeat yourself - make it work once and then reuse it
- If something gets too long or complex, break it up into logical chunks and put those chunks in functions
- Make a library of common stuff that you do over and over - perhaps share this with your friends...