Crash Course in Python; Unit 1

Procedural Python

UNIVERSITY OF CALGARY

Learning Outcomes

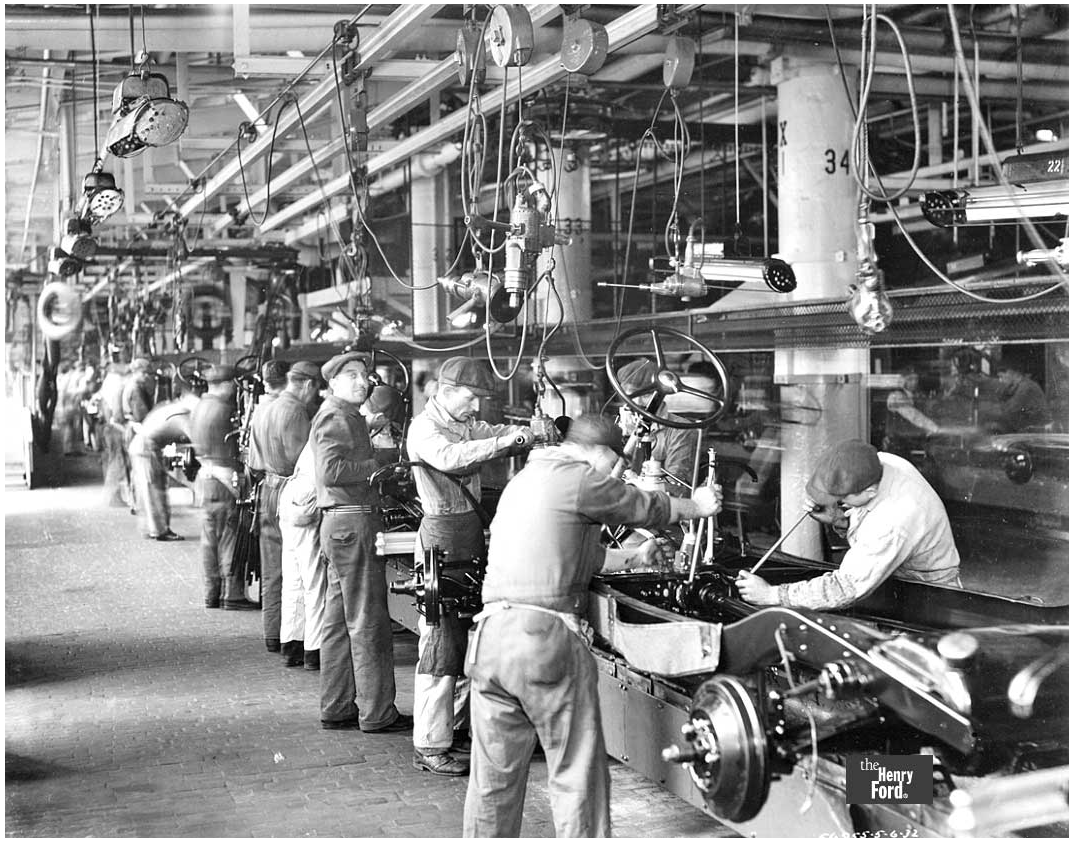
By the end of the module, you will be able to:

- Create re-usable blocks of code
- Create code which can change how it runs during runtime
- Devise code capable of iterating through collections of data
- Terminate these iterations conditionally
- Create custom collections of data within Python

All Python code described and used in this slideshow will be provided in a Jupyter Notebook file, which you can run on TALC to further experiment with. A (non-runnable) PDF version of said code will also be available for reference.

What Does it Mean to be Procedural?

- Procedural processes are those which work through a list of instructions (procedures) in a specified order
- Each of these process can update the state of the system, allowing for future instruction to react to and revise their procedures in response
- Most programming languages are procedural in some capacity, though exceptions (like Haskell) do exist



https://theoldmotor.com/wp-content/uploads/2016/02/1932-Ford-Chasiss-Assembly-Line.jpg

Procedural Programming Terminology, Part 1

- *Statement:* An instruction representing an action to be taken by the computer (such as '`print`'). They are often used alongside *arguments,* and statements can use and reference other statements
- *Argument:* A value passed to a statement when it is *called.* For example, with the '`print`' statement, the string we ask it to print is its only *argument*
- *Call:* A request to run a *statement* with a set of defined *arguments*

Procedural Programming Terminology, Part 2

- *Block:* A group of *statements* which always run together, forming a chain of instructions which itself can be treated as as statement
  - In Python, these are denoted using indentation (via tabs or spaces) underneath the block-defining statement
- *Scope:* The current level of accessibility that a statement or variable has. Python has two categories of scope:
  - *Global*: Element defined globally can be accessed and modified anytime during the program
  - *Local*: Elements created locally within a *block* can only be accessed by other elements within the same *block.*

Flow Control

UNIVERSITY OF CALGARY

Python Flow Control

- Flow control allows a program to dynamically change how it functions
- Python has three main forms of flow control:
  - Functions: Named blocks of code which can be re-used as needed
  - Conditionals: Blocks of code bound to only run under certain conditions
  - Loops: Blocks of code which will cycle repeatedly until some condition is met

https://momentum.vicc.org/wp-content/uploads/2011/06/decisiontree.jpg

Functions

- Defined in Python with the ‘`def`’ keyword:

```
~ def test(arg1, arg2):
~   print(f"{arg1} - {arg2}")
```

- The name of a function can be anything which abides by variable naming rules
- Functions can have one, many, or no arguments
  - Each argument’s names must also abide by the same variable naming rules
- Code placed within the function must be indented by some white space (usually a tab or set of spaces) to be recognized as part of that function
  - Any code within the function can use the values passed in as arguments, treating them as variables that share their name

Declaring and Calling a Function

- The following is a *function declaration*, which *declares* a function named "`double_val`", which takes one argument (`val`) and doubles it, reporting the result

```
~ def double_val(val):
~   ret_val = val * 2
~   print(ret_val)
```

- Now that the function has been , it can be *called* by its name to run, allowing us to re-use the code within the function alongside different values:

```
~ double_val(2)
> 4
~ double_val(6)
> 12
```

Variable Access and Functions

- Variables declared in the *global* scope can be accessed within a function, but variables declared within a function are *local* within it and cannot be accessed outside of it!

```
~ foo = 4
~ def bar(baz):
~   # This does not raise an error; ‘foo’ is globally scoped
~   bing = baz * foo
~   print(bing)
~ bar(3)
> 12
~ # This will raise an error; 'bing' is locally scoped in 'bar' and
~ # cannot be accessed outside of it
~ print(bing)
> NameError     Traceback (most recent call last)
> ...
> NameError: name 'bing' is not defined
```

Returning Data from a Function

- We can return local-scope data from a function using the '`return`' statement, allowing data generated within a function to be accessed outside of it

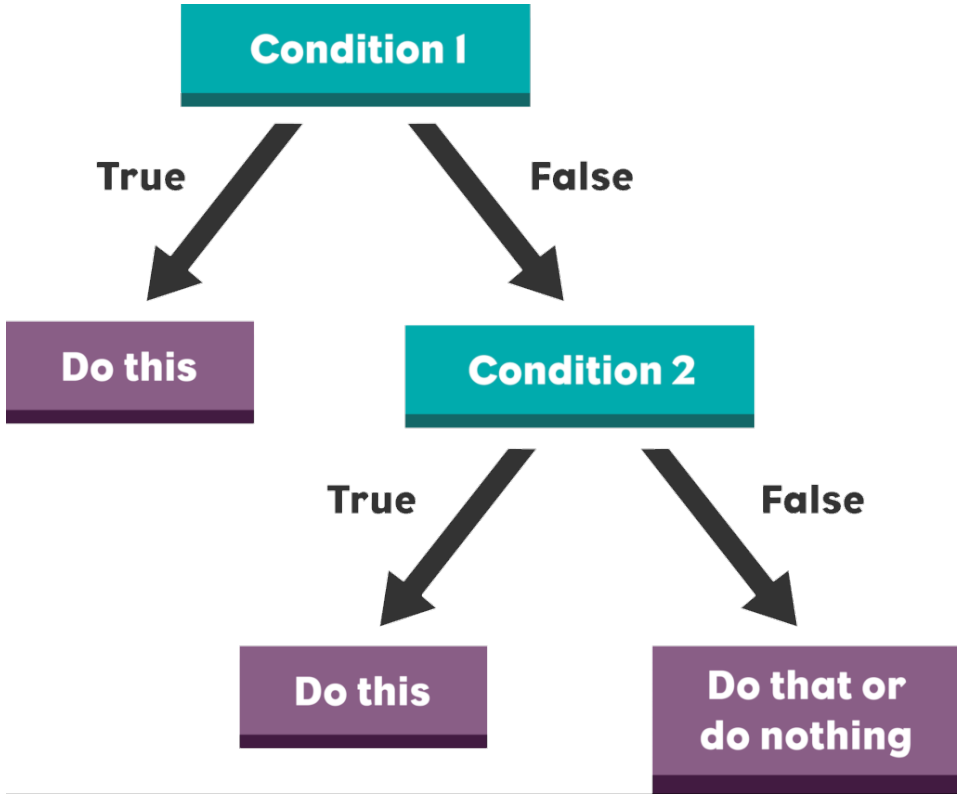
```
~ def triple_plus_two(val):
~   result = val * 3
~   result += 2
~   return result
~ new_val = triple_plus_two(4)
~ print(new_val)
> 14
```

- These returned value(s) can be immediately passed to another function as well

```
~ def minus_3(val):
~   return val - 3
~ val = minus_3(triple_plus_two(2))
~ print(val)
> 5
```

Conditionals

- Allows blocks of code to be run conditionally, as specified using either a boolean (true/false) value or statements which return a boolean value
- Python has three conditional statements:
  - '`if`': Runs the code within its block if the statement its bound to is '`true`'
  - '`elif`': Runs if the boolean statement provided evaluates to true *and* the prior '`if`' or '`elif`' statement failed to run
  - '`else`': Runs if the prior '`if`' or '`elif`' statement failed



https://technovationchallenge.org/wp-content/uploads/2019/03/if-else-if-chart.png

An Example Conditional

- Below is a simple if-elif-else chain, which will check and report whether two entered values are greater than, less than, or equal to one another:

```
~ def compare_values():
~   val1 = input("Enter a number:")
~   val2 = input("Enter another number:")
~   if val1 > val2:
~     print("The first value is greater than the second")
~   elif val1 < val2:
~     print("The first value is less than the second")
~   else:
~     print("The two values are equal")
~ compare_values()
> Enter a number: 4
> Enter another number: 5
> The first value is less than the second
```

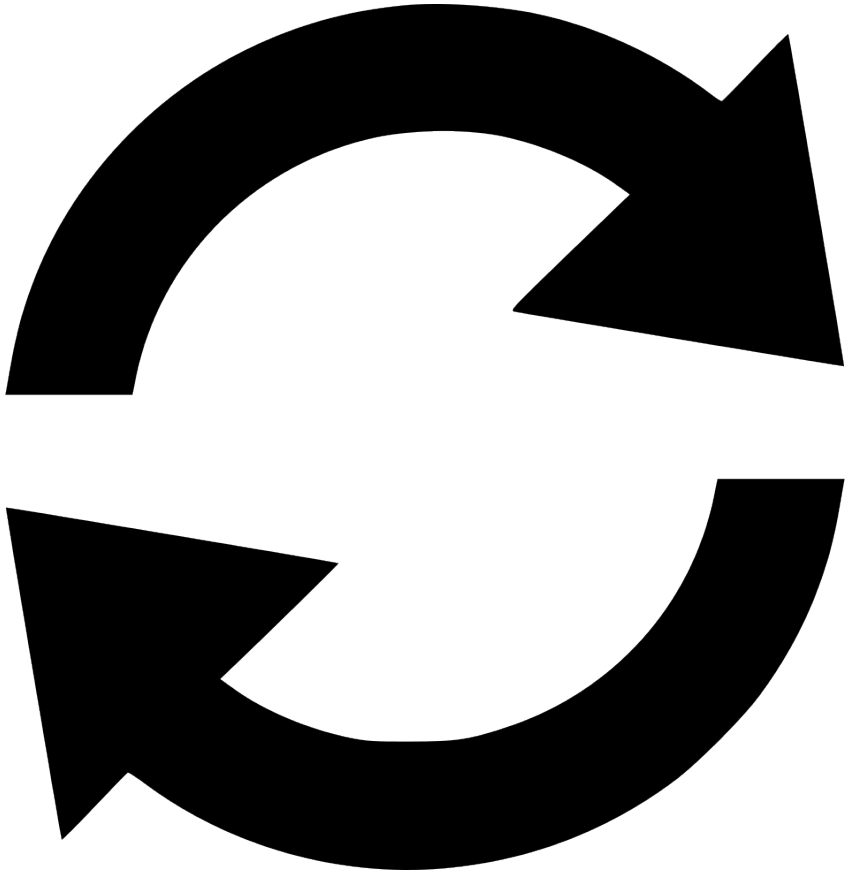
Nested Conditionals

- Conditionals can be nested within one another to form branching decision structures:

```
~ def evaluate_program_language(value):
~   if value == "Python":
~     print("Good choice!")
~   elif value == "R":
~     response = input("Are you a statistician?")
~     # lower() just forces the response to lower case
~     if response.lower() == "yes":
~       print("Reasonable choice")
~     else:
~       print("Bad choice")
~ evaluate_program_language("Python")
> Good choice!
~ evaluate_program_language("R")
> Are you a statistician? No
> Bad choice
```

Loops

- Structures which repeat a block of code until some condition is met
- Python has two loop structures:
  - '`while`': repeats the block of code until an if-like statement fails to be met. *This can result in the loop repeating indefinitely*
  - '`for`': repeats the block of code once for every element within an *iterable* (an object which can be *iterated* through)



https://www.onlinewebfonts.com/icon/477127

‘While’ Loops

- Like an ‘`if`’ statement, ‘`while`’ statements require a single statement which evaluates to a boolean value (true/false). This condition is evaluated once every loop, before the loop’s code block is run.
  - This means the block of code can also never run, if the condition is false when the ‘`while`’ statement is encountered

```
~ i = 0
~ while i < 5:
~   print(i)
~   i += 1
> 0
> 1
> 2
> 3
> 4
```

‘For’ Loops

- For loops use an *iterable* to determine when to terminate, running their block of code once per iteration. The element being used within the *iterable* for each iteration can be labelled as a variable, being treated like a function’s argument:

```
~ # Note: ‘range’ provides the range of numbers
~ # between 0 and the specified value
~ for i in range(5):
~   print(i)
> 0
> 1
> 2
> 3
> 4
```

Terminating Loops Prematurely

- Loops can also be terminated explicitly using the '`break`' statement:

```
~ for i in range(5):
~   if i > 1:
~     break
~   print(i)
> 0
> 1
```

- Alternatively, a single cycle (rather than the entire loop) can be skipped using the '`continue`' statement:

```
~ for i in range(3):
~   if i == 1:
~     continue
~   print(i)
> 0
> 2
```

Lists

- Lists are collections of data which can be iterated through in order, and are declared using square brackets.
- Lists are *iterable*, and thus can be used as part of a '`for`' loop:

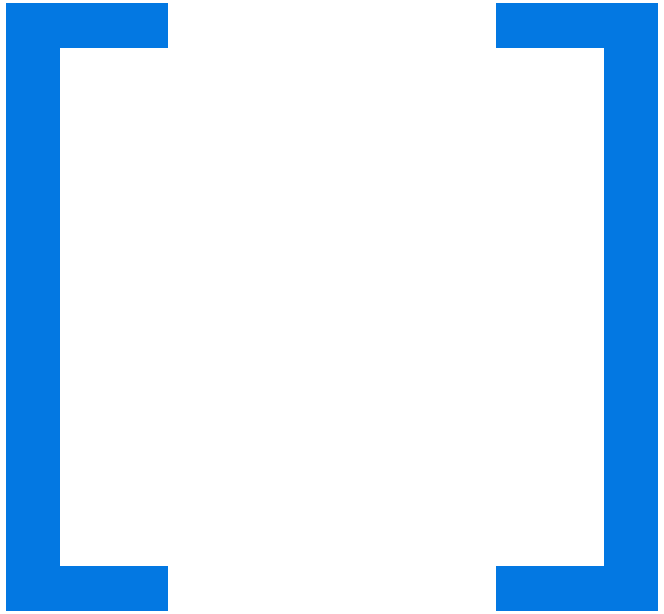
```
~ list1 = [1, True, 'F', "Hello World!"]
~ for i in list1:
~   print(i)
> 1
> True
> F
> Hello World!
```

http://4.bp.blogspot.com/-8HfLPbR1n0M/VLREeYjL0pI/AAAAAAAAAb8/CxNo7Qn3vG0/s1600/To-Do%2BList.png

Selecting List Elements

- Lists are queried identically to strings. This is because strings are actually just collections of characters, and both use the same querying syntax:

```
~ list1 = [1, 4, 9, 16]
~ print(list1[0])
> 1
~ list2 = list1[2:]
~ print(list2)
> [9, 16]
~ print(list1[1:3])
> [4, 9]
```



Updating Existing Lists

- To add an element to an existing list, we can use the ‘append’ function of the list:

```
~ list1 = [‘a’]
~ list1.append(‘b’)
~ print(list1)
> [‘a’, ‘b’]
```

- Note that appending a list to another will append the (new) list itself, not its contents. If you want to extent a list with another’s contents, use the ‘extend’ list function:

```
~ list1 = [‘a’]
~ list2 = [‘b’]
~ list1.append(list2)
~ print(list1)
> [‘a’, [‘b’]]
~ list1 = [‘a’]
~ list1.extend(list2)
~ print(list1)
~ [‘a’, ‘b’]
```

Removing Elements from Lists

- We can delete an element based on its location using the ‘pop’ function. If we do not pass an index, it will always delete the last element in the list instead:

```
~ num_list = [0, 2, 4]
~ num_list.pop(1)
~ print(num_list)
> [0, 4]
~ num_list.pop()
~ print(num_list)
> [0]
```

- Alternatively, an element can be removed based on it’s value with the ‘remove’ function. This will only delete the first instance of said element, however:

```
~ char_list = [‘A’, ‘B’, ‘C’, ‘B’]
~ char_list.remove(‘B’)
~ print(char_list)
> [‘A’, ‘C’, ‘B’]
```

Putting it All Together!

UNIVERSITY OF CALGARY

Lets Test your Learning!

- You have been tasked to create another Python program. This one should:
  - Greet the user
  - Request numbers from the user until:
    - The sum of all collected *absolute* values exceeds 100, or...
    - The user enters “done”
  - Using the set of numbers collected prior, report the following for each:
    - The number, plus 2
    - The cumulative product of all numbers up to and including this new number.
      - Said cumulative product should be initialized with a value of 1

An incomplete Python script file has been provided with comment hints for each of the above steps for you to test your learning. A key for this document has also been provided, should you want to check your answer

Thank you!

Please send inquiries to kalum.ost@ucalgary.ca

UNIVERSITY OF CALGARY