Crash Course in Python; Unit 1

Procedural Python

UNIVERSITY OF CALGARY

Learning Outcomes

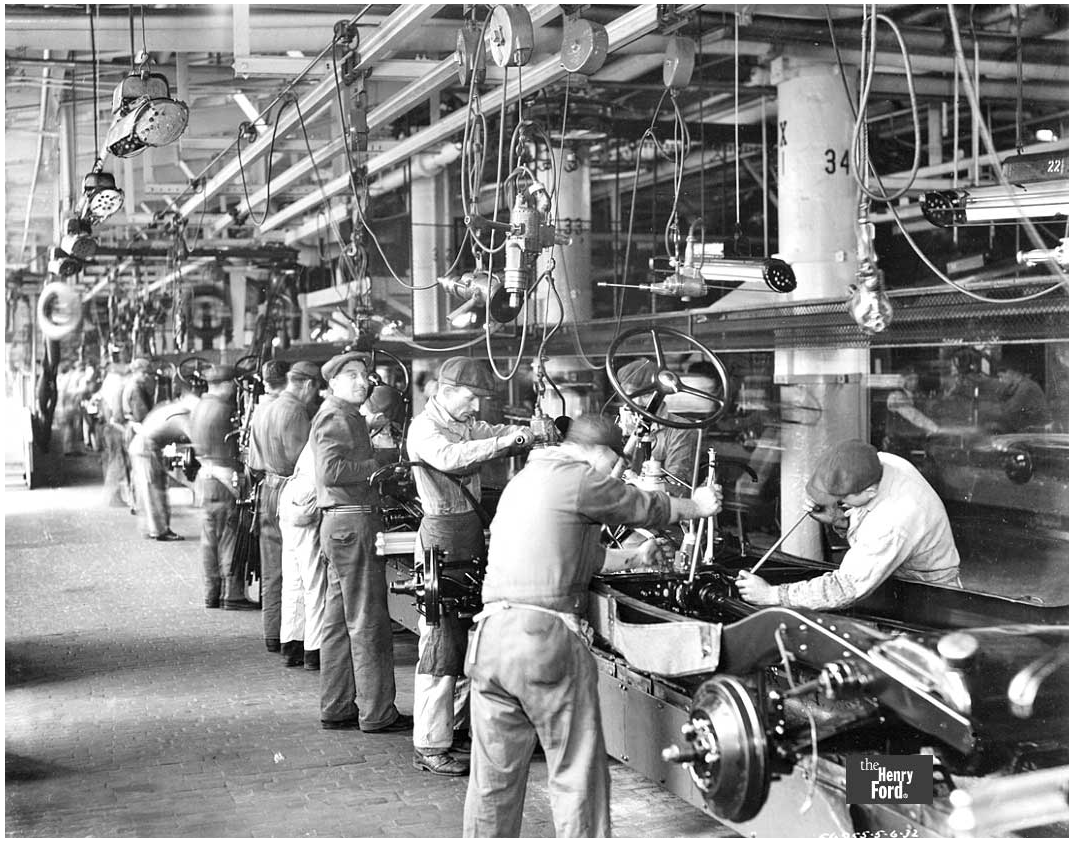
By the end of the module, you will be able to:

- Create collections of data within Python
- Create Python scripts which change how they run in during runtime
- Devise code capable of iterating through collections of data, with early termination conditions
- Iterate line by line through a input file programmatically, reporting on the files contents as you do so

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 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 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, often with *arguments*
  - Can themselves use and reference other statements through *calls*
- *Argument:* Values passed to a statement when it is *called*
- *Call:* A request to run a statement, alongside any *arguments* required for the to-be-taken action

Procedural Programming Terminology, Part 2

- *Block:* A group of *statements* which always run together, effectively forming a composite *statement*
- *Scope:* The current accessibility that a statement or variable has. Generally split into two categories:
  - *Global*: Anything within global scope can be accessed and modified anytime during the script
  - *Local*: Elements within this scope can only be used and accessed by other elements within the same *block*, effectively being deleted when the block finishes its set of instructions.

Flow Control

UNIVERSITY OF CALGARY

Python Flow Control

- What allows a program to make decisions on what to do
- Python has three main forms of flow control:
  - Functions: Named blocks of code which can be re-used multiple times
  - 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 like so:

```
~ def name(arg1, arg2...):
~    # code to run
```

- The name of the function can be anything which abides by variable naming rules
- A function can have one, many, or no arguments
  - Each argument's name 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 this function can use the values passed in as arguments by using their name's, identically to variables outside of a function

Declaring and Calling a Function

- Consider the following function *declaration*, which simply doubles a passed value and reports the result

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

- Once declared, this function can be *called* by its name to run, allowing us to re-use the code within the function alongside different values as needed:

```
~ double2 = double(2)
> 4
~ double6 = double(6)
> 12
```

Variable Access and Functions

- Variables declared in the *global* scope can be accessed within a function, but variables declared within a function 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, as ‘bing’ is locally scoped within ‘bar’!
~ print(bing)
> NameError     Traceback (most recent call last)
> ----> 1 print(bing)
> NameError: name 'bing' is not defined
```

Returning Data

- We can return a result from a function using the ‘return’ statement, allowing data generated within a function to be accessed outside of it

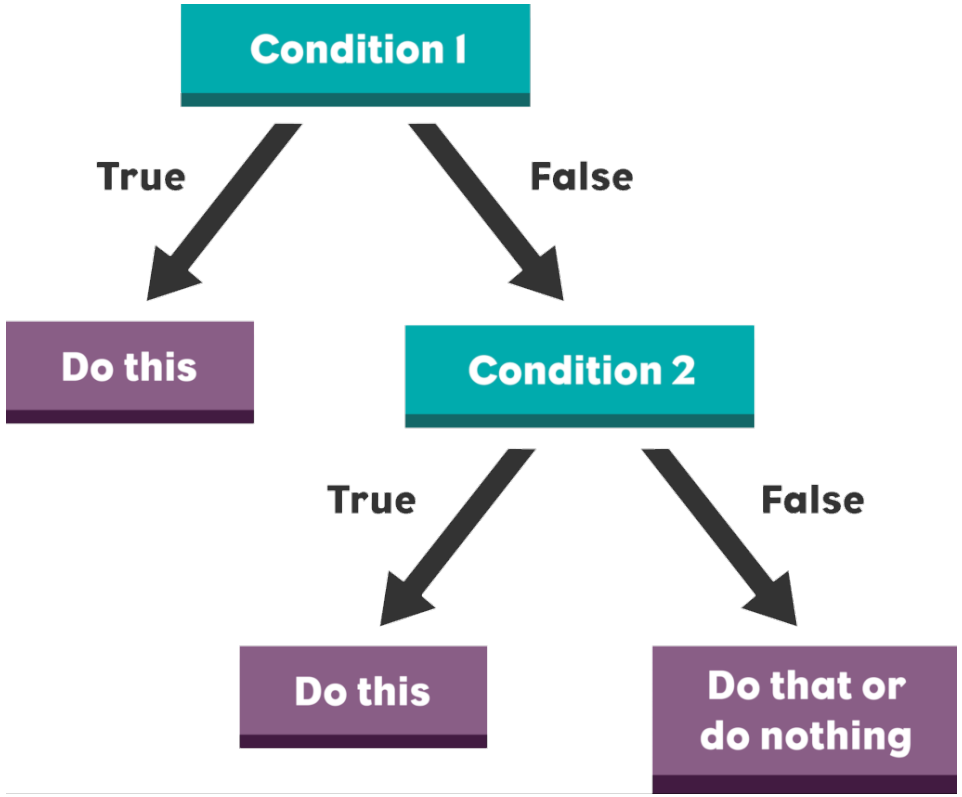
```
~ def triple_plus_two(val):
~   result = val * 3
~   result += 2
~   return result
~ val = triple_plus_two(4)
~ print(val)
> 14
```

- 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 boolean (true/false) values or statements which return or evaluate to boolean values
- Python has three types of conditional statements which we can use to conditionally run code:
  - '`if`': Runs the code if the statement provided is true
  - '`elif`': Runs if the boolean statement provided evaluates to true *and* the prior 'if' or 'elif' statement did not run its code
  - '`else`': Runs if the prior 'if' or 'elif' statement did not run its code



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

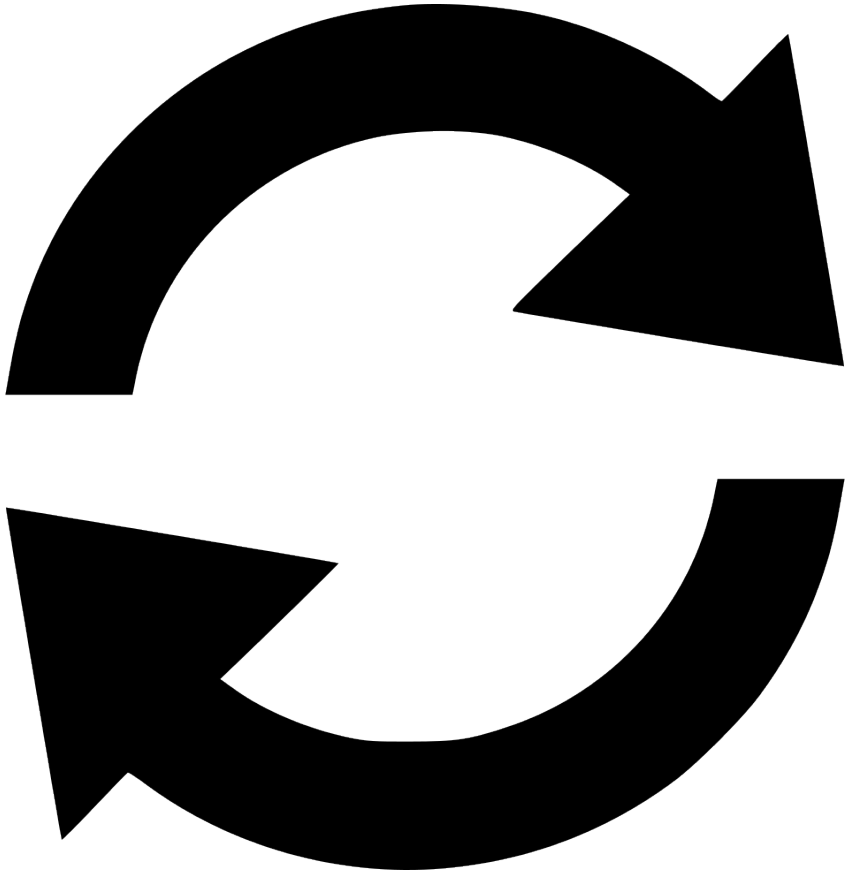
Nested Conditionals

- 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?")
~     if boolean(response):
~       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

- Repeats a block of code until some conditions is met
- Python has two loop structures:
  - '`while`': repeats the block of code until a if-like fails to be met. *This can result in a loop repeating infinitely.*
  - '`for`': repeats the block of code once for every element within an *iterable* (often a range of numbers or set of to-be-analyzed values)



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

‘While’ Loops

- Like ‘`if`’ statements, ‘`while`’ statements require a single statement which evaluates to a boolean value (true/false).
  - Said statement is evaluated once every loop, before the loop’s code block begins to be 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 val < 5:
~   print(i)
~   i += 1
> 0
> 1
> 2
> 3
> 4
```

‘For’ Loops

- For loops, in contrast, require an *iterable* (any object which can be *iterated* through) to be provided, and will run their block of code once per iteration.
  - The element being used within the *iterable* can be labelled like a variable and accessed within the code block, similar to a function 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, bypassing their condition/iterable, using the ‘break’ statement:

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

- Alternatively, a single iteration (but not the entire loop) can be skipped using the ‘continue’ statement:

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

A Special Iterable; ‘Lists’

- Lists are a very common type of iterable in Python, representing collections of data that is arbitrary in form by explicitly defined by the code in some way
- Lists containing defined values can be defined using square brackets, and (being iterable) can also be used as part of for loops:

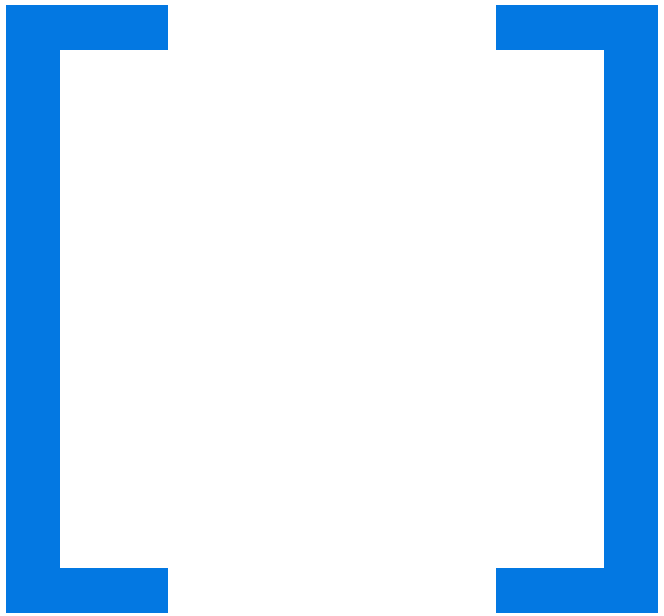
```
~ list1 = [1, True, ‘F’, "Hello World!"]
~ for i in list1:
~   print(i)
> 1
> True
> F
> Hello World!
```

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

Selecting Elements from Lists

- Strings are also *iterable*; as a result, Python is designed to allow the same same querying tools used with strings to be used on lists (and other *iterables* with defined contents):

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



Updating Existing Lists

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

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

  - Note: Appending another list will append the list itself, not its contents. If you want to do so, 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 it's 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()
> [0]
```

- Alternatively, and 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 again to create a Python program can:
  - 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