Lesson 1: Welcome to Fundamentals I

Table of Contents

1.1. Objectives

- Examine the implications of using computation to solve a problem
  - Discuss best practices for using computation to solve a problem
  - Suggest types of problems that can be solved through computation

    - Show how computation can solve a problem

- Recognize key computer science concepts
    - Identify data types used in Python scripting
    - Identify data structures used in Python scripting
    - Define variables and strings
    - Recognize how queries operate

- Demonstrate the ability to build basic scripts using Python scripting language
    - Use various data types and structures in Python scripting
    - Collect data using Python scripting
    - Extract data using Python scripting
    - Develop advanced data structures using Python scripting

---

1.2. Overview

The following lesson is broken up into the following parts:

- Course Overview
- Additional Assistance
- Computational Thinking
- Python
- Using Jupyter Notebook
- Practice Python

1.3. Course Overview

This is an instructor-led course designed to help shift how you think about solving problems. Its intent is to show you the potential application of coding as a tool to solve problems. This course will not turn you into a computer scientist, data scientist, or Silicon Valley CEO in 40 hours of classroom time. The emphasis is data-related learning objectives that can help you meet the mission with greater speed and precision.

1.4. Additional Assistance

Type	When
Open Lab with Instructors	Monday - Thursday, 1500-1600
Community of Practice (CoP)	Anytime, anywhere

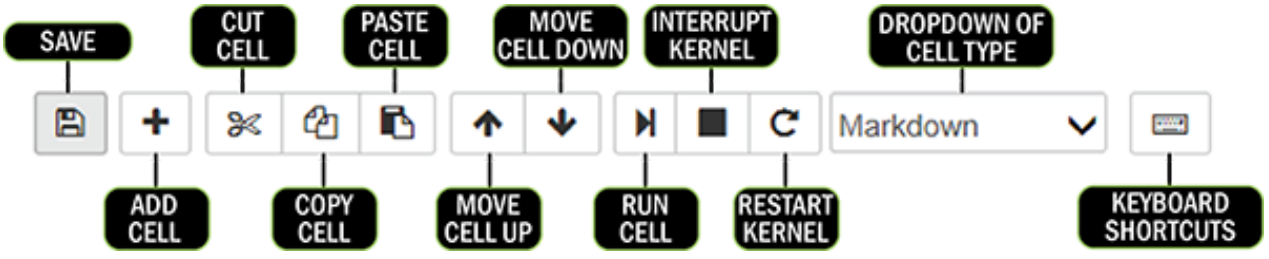
There are multiple services and groups available upstairs that can help you with your coding development by providing additional resources, skills, and even desk-side assistance.

Title	Service	Contact
SBU	Access to Google, StackExchange, etc.	NA
Python Party	Community of coders who share their coding success stories and answer questions from the coding community.	https://rspace.dodiis.ic.gov/rspace/groups/python-party (https://rspace.dodiis.ic.gov/rspace/groups/python-party)
OHI	Desk-side assistance with coding and assistance with JupyterHub.	https://intellipedia.intelink.ic.gov/wiki/NGA%27s_Office_Hours_Initiative (https://intellipedia.intelink.ic.gov/wiki/NGA%27s_Office_Hours_Initiative)
ADV	Developer tools (including Python documentation), stack exchange mirror, and a highside PyPI repository, among other resources.	TBA

Instructor Guidance: Discuss days and times for lab hours (with and without instructors). Write email addresses and phone numbers on the board as appropriate.

1.5. Jupyter Notebook Basics

As shown below, Jupyter Notebooks consist of a number of code or Markdown cells. Set the type of any cell with the drop-down on the right end of the toolbar. Markdown allows you to write notes and text in various forms such as lists or tables. The Jupyter Notebook User's Manual is a great resource to learning more details about Markdown formatting.



Instructor Guidance: Explain the concept of a cell and the benefits of having code broken into sections.

Exercise: Use the "Run Cell" button to run the cell below.

In [ ]:

```
1111 * 1111
```

Exercise: Add a cell below this one using the "Add Cell" button.

Exercise: Copy the cell you just added and paste it below this cell.

In [ ]:

```
# Exercise: What is the type of this cell?
```

Exercise: What is the type of this cell?

1.6. Python

References:

- Wikipedia: Python (programming language) (https://en.wikipedia.org/wiki/Python_%28programming_language%29)

Python is a widely used programming language for general-purpose programming. Python was created by Guido van Rossum and first released in 1991. Python has a design philosophy that emphasizes code readability (notably using whitespace indentation to delimit code blocks rather than curly brackets or keywords), and a syntax that allows programmers to express concepts in fewer lines of code than might be used in languages such as C++ or Java. Van Rossum is Python's principal author, and his role in deciding the direction of Python is reflected in the title given to him by the Python community: Benevolent Dictator For Life (BDFL).

Python is named after Monty Python's Flying Circus (https://en.wikipedia.org/wiki/Monty_Python%27s_Flying_Circus). The core philosophy of the language is summarized by the document The Zen of Python (https://en.wikipedia.org/wiki/Zen_of_Python), which includes aphorisms such as:

- Beautiful is better than ugly
- Explicit is better than implicit
- Simple is better than complex
- Complex is better than complicated
- Readability counts

Instructor Guidance: Students can read the whole list of aphorisms in The Zen of Python by running `import this`

The PEP8 Style Guide for Python (https://www.python.org/dev/peps/pep-0008/) is a fairly short web document. Although your Python code is not being graded for style in this course, it is a good practice to adhere to the Python Style Guide if only to increase the readability of your code by others, especially if you are seeking their help to solve a problem.

Some Python style highlights:

- **Indentation** - Use four spaces per indentation level (Tabs are set to four spaces in Jupyter Notebooks)
- **Naming Variables** - lower_case_with_underscores
  - *Function and Variable Names* - Names should be lowercase, with words separated by underscores as necessary to improve readability.
  - *Constants* - Constants are usually defined on a module level and written in all capital letters with underscores separating words. Examples include PLANCKS_CONSTANT and PI.

Regardless of whether you adhere to a style guide or not, try to be consistent in your code.

1.6.1. Python 3

In the classroom, we will be using Python 3. Some of you who've used Python before might have used older versions. There are slight differences between versions, but we will stay consistent throughout the course with version 3, and point out some major differences between Python 2 and Python 3 along the way.

In [ ]:

```
import sys
sys.version
```

1.6.2. Some Vocabulary for Python

Here is list of some common terms and concepts that you will hear in this course.

- **Syntax** - Syntax refers to how you write code, such as spacing, punctuation, and indentation. Python depends on correct syntax to properly read and execute the code you write. A single mistake in syntax will cause Python to execute the code differently than you intended or to throw an error, meaning Python will stop running your code and then give you an error message describing what went wrong.

  > REMEMBER: Computers are not smart; they are just really fast. Python syntax is designed to ensure there is no ambiguity in your instructions to the computer.

- **Variable** - A variable name is a stand-in for some data value. Think back to when you learned to use "x" as a stand-in for numbers in math class. Variables in code work just like this, but are more powerful. A variable name in Python can be any letter, word, or series of words linked by underscores (_). Although you can use almost any word for variable names, the best practice is to be *descriptive*. Variables stand in not just for numbers, but also for text, Boolean values, lists of things, or any other type of data. To assign a data value to a variable in Python, we use a single equals sign, like `x = 4` or `some_text = 'Hello world!'`.

- **Comment** - A comment is text that is ignored by the computer. It is a description that provides more information about the thought process or logic that the code is representing. Any text that follows a pound

sign (`#`) in a line of code will be considered a comment, and will not be evaluated by the computer. This allows us to leave notes for ourselves and other developers to reference.

Exercise: In the code cell below, the variable `y` is assigned to the integer `4`. Create a new variable `z` and assign it the integer `3`. Then, add `y + z` and run the code you have written.

In [ ]:

```
y = 4
```

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
z = 3

y + z
```

- **Function** or **Method** - Think of functions and methods as little programs or apps. They are bodies of code that take specified input(s), do some actions (even running other functions), and then usually provide some output. You can think of every computer program you use (i.e., Microsoft Word, Internet Explorer, Super Mario, etc.) as a group of functions that work together to carry out all of the things the program does. You will be writing some simple functions in this course before too long.

  > NOTE: The name of a Python function is always followed by parentheses; it's the easiest way to identify a function in code. The input(s) for a function or method go inside of those parentheses.

In the code cell below, we call the `print()` function. Run the code in the cell below to see what `print()` does.

In [ ]:

```
print('Hello world!')
```

- **Arguments** or **Parameters** - These are just the technical terms for the inputs that a function or method takes. Some functions and methods require certain input(s), while others require no inputs. In the code cell above `'Hello world!'` is the argument for `print()`.

- **Library** - A library is a collection of code that defines certain methods and may also define the nature of special data types, like `datetime` objects made to calculate or track dates and times. When you need to use a particular method or special data type, you just import that library into the Python script you are

writing. Where do libraries come from? They are written by programmers and shared with others, usually online.

1.6.3. Guided Exercise: Finding the Square Root

In the cell below, try to use the square root function, `sqrt()`, from the math library to find the square root of a number.

In [ ]:

```
sqrt(4)
```

Congratulations on your first of many error messages! Error messages are friendly. They tell you what went wrong so you know what to fix in your code. Error messages won't harm or break your script, but Python will stop executing code when it throws an error message. The `NameError` tells us that Python does not know what `sqrt()` is. Why? Because `sqrt()` is from the math library and we did not import that library yet. To use `sqrt()`, we just need to import the Math Library using a special command: `import`.

In [ ]:

```
## Run this code by holding Ctrl and hitting Enter ##
import math

math.sqrt(4)
```

Now `sqrt()` is available inside our script, but notice you have to type it as `math.sqrt()` because even though we imported the Math Library we still have to tell Python where to find the `sqrt()` method. This syntax of `library_name.method_name()` is what you will use when you want to use methods from libraries.

Also, notice that the word `import` shows up bold green in the code cell above while the word `math` is in normal black text. This is because `import` is a reserved word, or keyword, in Python. Reserved words are words that are already assigned some special purpose and cannot be used for other things, like variable names or function names. You will learn more reserved words and what they do throughout this course.

Instructor Guidance: Point out how an import only needs to occur once per program. Once imported, the library is available until a kernel restart. Briefly mention the number next to the cell indicating the order in which you ran the cells.

1.6.4. Python Resources

Online Resources

- Python.org (https://www.python.org/)
- Python for Non-programmers (https://wiki.python.org/moin/BeginnersGuide/NonProgrammers)
- The Hitchhikers Guide to Python (http://docs.python-guide.org/en/latest/)
- PEP8 Style Guide for Python (https://www.python.org/dev/peps/pep-0008/)
- O'Reilly's Think Python: How To Think Like a Computer Scientist, 2nd Edition (http://greenteapress.com/thinkpython2/thinkpython2.pdf) - Allen Downey made this great introduction to Python 2.7 available for free at Green Tea Press. Should you decide to do the practical exercises in the book, Mr. Downey has made the solutions (http://greenteapress.com/thinkpython/code/) available as Jupyter Notebooks.
- Automate the Boring Stuff with Python - Practical Programming for Total Beginners (https://automatetheboringstuff.com/) - Written for office workers, students, administrators, and anyone who uses a computer to learn how to code small, practical programs to automate tasks on their computer.
- StackOverflow (https://stackoverflow.com/questions/tagged/python) - A great resource to find solutions for any coding problem you might have in this course and in your jobs. If you find yourself stuck with a problem during the practical exercises, think about the problem you are trying to solve and ask Google. You will most likely find that you are not the first to have the problem and there are plenty of people out in the StackOverflow community who can help.
- Jupyter Notebook (software) (https://jupyter.org/) - The classroom exercises will be conducted using Jupyter Notebooks. Jupyter Notebooks, or iPython Notebooks as they were called previously, are a great way to create code, add documentation, and share both the code and the results with others.
- What is Jupyter Notebook (YouTube) (https://video.search.yahoo.com/search/video;_ylt=A0LEVxT2upRZU94AX1tXNyoA;_ylu=X3oDMTEyNGJk
p=jupyter+notebook+tutorial&fr=dss_yset_chr#id=8&vid=74d3ef639eeabb387d6267364dab002b&action=v
- Jupyter Notebook Users Manual (https://athena.brynmawr.edu/jupyter/hub/dblank/public/Jupyter%20Notebook%20Users%20Manual.ipynb)
- Cheat Sheets on the CSCI2011 course website
- CSCI2011 Course Blog/Parking Lot

Offline Resources

- Python's `help()` function

1.6.5. Reserved Keywords and Built-In Functions

1.6.5.1. Keywords

References:

- Python: Keywords (https://docs.python.org/3.4/library/keyword.html)

Reserved words, or keywords, are words within Python that have specific use cases and cannot be used as variable names. They show up as green and bold inside Jupyter Notebook code cells. The keyword library has a list of all Python's keywords. Run the cell below to print them out.

In [ ]:

```
import keyword

print(keyword.kwlist)
```

All the above words are not usable as variable names or for any other purpose than their specific functions. However, remember that Python is case sensitive. It is possible, for instance, to use `Elif` instead of `elif` as a variable name, but it is generally considered bad practice.

1.6.5.2. Built-In Functions

You will also notice that some words show up as light green (as opposed to bold green) when you type them into a code cell. These are built-in functions (i.e., functions that come prepackaged for you with your version of Python).

Here are some examples of built-in functions.

```
print()
type()
max()
```

Unlike reserved words, Python will let you re-assign built-in function names to other values, *but you should not do this*. Once you re-assign a function's name to something else, you lose the ability to use that function, which is bad because these functions are fundamental to some core Python capabilities. Never do something like the following. If you're ever creating a variable and the variable name turns green, choose a different name.

Do not do this:

```
print = 'message'
list = [1, 2, 3]
```

1.6.6. Python's `help()` Function

Python includes an interactive `help()` function, which initializes a help menu. If an argument is passed into the `help()` function (such as `math`), the documentation for that topic will be printed. The `help()` function works with imported libraries as well as keywords and built-in functions.

Instructor Guidance: Have them type "math" at the `input()` prompt for the first choice. After that,

participants should make their own choices. When done, just type "quit" or hit Enter with an empty dialog box.

In [ ]:

```
help()
```

In [ ]:

```
help('keywords')
```

1.7. Jupyter Notebook Tips and Tricks

1.7.1. Keyboard Shortcuts

These notebooks can be very helpful as you navigate this course. Let's check out some of the functionality built into Jupyter Notebooks. There are a number of keyboard shortcuts that work in Jupyter Notebook. You can find a complete list of shortcuts by clicking the keyboard button in the menu above. The following three shortcuts are very useful because they allow you to run, or execute, cells.

Keyboard Shortcuts for All Modes

Shortcut	Description
Shift + Enter	Execute the contents of the cell, and move onto the next cell
Ctrl + Enter	Execute the contents of the cell, but stay with this cell
Alt + Enter	Execute the contents of the cell, insert a new cell below

Exercise: Run the code cell below using a keyboard shortcut.

In [ ]:

```
print('Good job!')
```

When you're using Jupyter Notebook, you're operating in one of two modes: *Edit Mode* (green cell margin) and *Command Mode* (blue cell margin). Edit Mode allows you to change the contents of a cell, whereas Command Mode allows you to perform operations on a cell or group of cells as a whole, such as running, copying, or deleting. To enter Edit Mode, either click inside the cell or hit Enter when the cell is highlighted. To enter Command Mode, either click outside of the cell or hit Escape.

Keyboard Shortcuts for Switching Between Modes

Shortcut	Description
Enter	enter Edit Mode
Esc	enter Command Mode

Jupyter also supports *Tab Completion*. This feature will complete certain code snippets for you, and is mostly used for completing variable and function names. Tab Completion is useful because it allows you to type code faster and prevents typos.

Exercise: In the code cell below, a variable is created for you. If you have to type this variable name with some frequency, that can get cumbersome. Start to type the variable name in the cell below, then hit Tab to complete it.

In [ ]:

```
very_long_variable_name = 'test string'
```

In [ ]:

```
## YOUR CODE GOES HERE ##
```

Keyboard Shortcuts for Command Mode

Shortcut	Description
m	Convert a cell to Markdown
y	Convert a cell to Code
l	Show line numbers in the selected cell
a	Insert a cell above the selected cell
b	Insert a cell below the selected cell
c	Copy the selected cell
x	Cut the selected cell
v	Paste cell below the selected cell
dd	Delete selected cell
z	Undo a cell deletion
Up Arrow	Highlight the cell above
Down Arrow	Highlight the cell below

Exercise: Change the cell type of the cell below. Once you change it to a code cell, try running it.

#Change the type of this cell from Markdown to Code and run the cell.

#You can do this by using the toolbar or the keyboard shortcuts. Try both.

print('Nice Job!')

Exercise: Delete one of the duplicated cells below.

This cell is in here twice, get rid of one of them

This cell is in here twice, get rid of one of them

Exercise: In case you accidentally delete a code or markdown cell you've been working hard on, you *can* undo it. Undo the deletion from above.

Exercise: Add a code cell below, copy and paste the following line, then run it:

```
print("Woah! You're getting the hang of this!")
```

Instructor Guidance: Poll the students for any other functionality they're interested in, or demonstrate some you feel is useful. They haven't had much experience at this point, so many may not know what to ask. Point out that this notebook can be used as a reference, so they can refer to it later as they'll continue to have access to this notebook.

Instructor Guidance: `Shift + Tab` can be used at the end of a variable name (without a period) to show current variable value and type. `Shift + Tab` inside of the parentheses of a function/method will reveal the function docstring.

Exercise: Try out Jupyter Notebook functionality.

In [ ]:

```
## Try some code ideas here ##
```

In [ ]:

```
## Try some code ideas here ##
```

In [ ]:

```
## Try some code ideas here ##
```

In [ ]:

```
## Try some code ideas here ##
```

If you want a more in-depth review of Jupyter Notebooks and how you can customize them, check out the excellent Jupyter Notebook Users Manual

(https://athena.brynmawr.edu/jupyter/hub/dblank/public/Jupyter%20Notebook%20Users%20Manual.ipynb).

1.7.2. The Kernel Menu

Under the hood, Jupyter Notebook has a Python *kernel* that stores the variables you've created and remembers what libraries you've imported. The kernel can only run one code cell at a time, but everything stored in the kernel is available anywhere inside your Jupyter Notebook. Splitting up code into different cells allows you to view outputs at every step. For example, the variable we create in the first cell below is still available to us in the second cell.

In [ ]:

```
string_variable = 'Test String'
```

In [ ]:

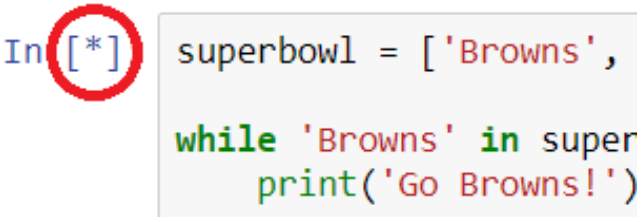
```
print(string_variable)
```

Kernels can break, and when they do we need to restart them. Let's break our kernel on purpose and walk through how to restart it. Run the code cell below.

In [ ]:

```
help()
```

Notice the `[*]` next to the cell. When you see an asterisk in the brackets next to a cell, it means that cell is running. Sometimes, this is because the code is taking a while to complete, but this cell is waiting for an input in the dialog box, and until it gets that input it will hang there, waiting.



When a cell is finished executing its code, you'll know because a number will appear in the brackets next to the cell. That number shows the order in which you ran the cells in this notebook. An empty set of brackets next to a code cell means it hasn't been run in the current kernel.

Now run the above cell again. Because we ran the cell while it was still waiting on user input, it will never stop. Now we have to restart the kernel.

Go to the "Kernel" dropdown at the top, and click "Restart and Clear Output". You'll see a popup; select "Restart and Clear Output" in the popup. Wait until the `[*]` disappears. Let's try to reference our variable from before.

In [ ]:

```
print(string_variable)
```

Restarting the kernel means that all the previously stored memory is lost. If we need to access variables we created earlier in the script, we'll need to scroll up and rerun the cells that initialize them.

Exercise: Re-run the code cell that initializes the `string_variable` above, then run the code cell directly above to print it.

1.7.3. Output vs. `print()`

In Jupyter Notebooks, there are two ways to display output:

1. Type what you want to display inside the `print()` function.
2. Type what you want to display as the last line in a code cell.

Both ways are demonstrated in the code cell below.

In [ ]:

```
text = 'A message'

print(text)

text
```

Instructor Guidance: `print()` vs Jupyter Output may not seem important to students at this point, but mention how they are both just displaying the values that are being passed to them--not storing/saving them anywhere. Refer back to this discrepancy when we learn about functions' return statements in Lesson 5.

However, if what you are trying to display is not on the last line of the code cell, it will not be displayed. `print()` , on the other hand, will display what you ask it to regardless of where the print statement is in the code cell.

In [ ]:

```
text = 'A message'

text

print(text)
```

In the cell above, we moved the last line of the previous code cell so that it is no longer the last line. We see the print statement still works as expected, but the other output has disappeared.

Instructor Guidance: Ask the students how Python is interpreting the 'text' line of code. Let them think about if it is Python or Jupyter Notebooks that knows to display the value from the last line of code in a cell.

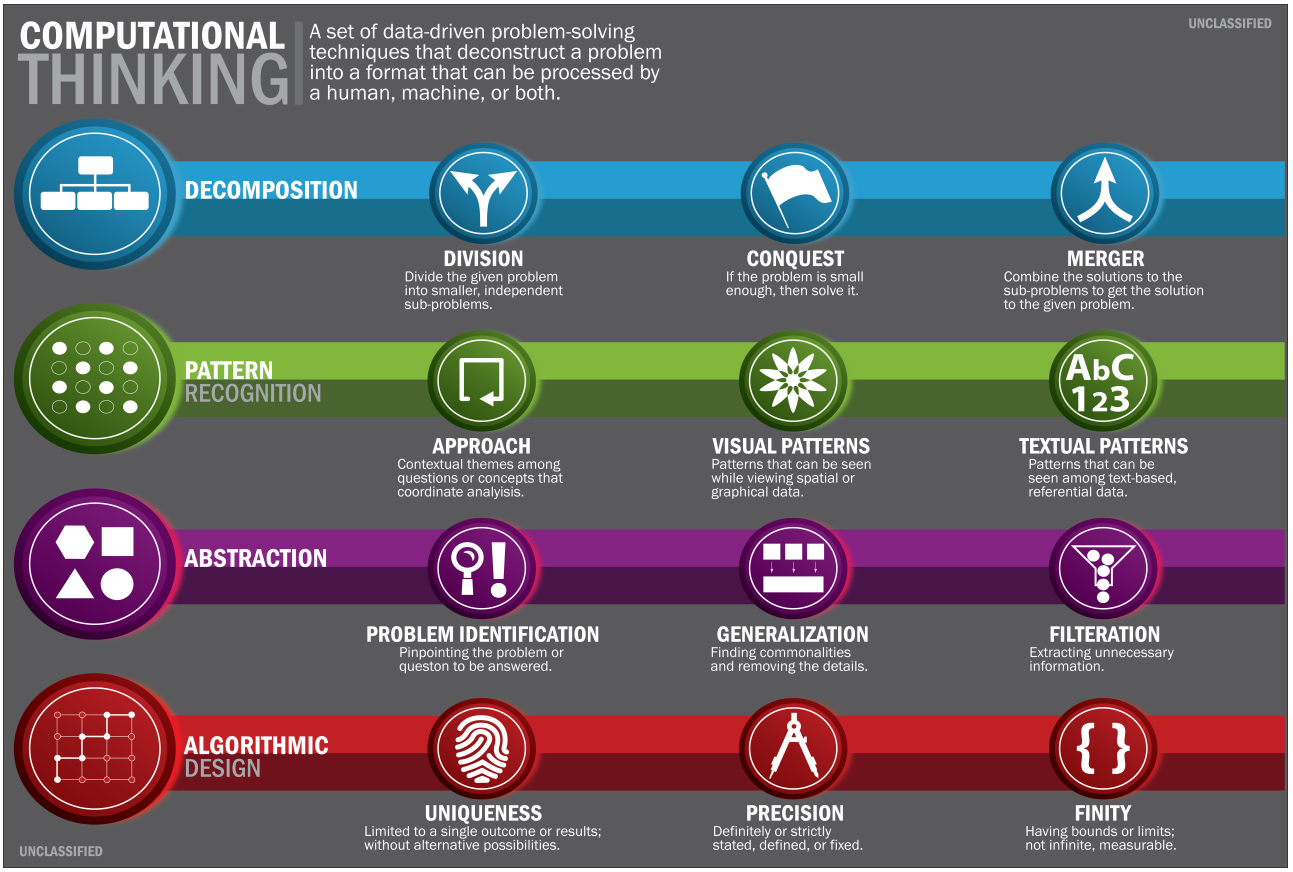
1.8. Computational Thinking

Computers are not smart, but they are fast. They need explicit instructions in order to solve a problem. This means that when we, as programmers, are presented with a problem, we have to provide the computer a complete set of instructions for it to do what we want. This requires a special mode of reasoning that we call *computational thinking*.

Instructor Guidance: Solving problems using Python (or any programming language) at NGA requires a clear understanding of computational thinking and how each component can be leveraged to further understand and answer problems. You will be expected to refer back to this section of Lesson

1 and address the four steps of computational thinking to solve problems as they relate to lessons, exercises, examples, student questions/comments, etc., throughout the course.

Instructor Guidance: Use the embedded computational thinking graphic taken from CMPT1000. Throughout CSCI2011, refer back to this graphic when discussing the process of thinking computationally.



The basic steps for approaching a problem with computational thinking are:

1. **Decomposition:** Breaking down data, processes, or problems into smaller, manageable parts
   A. Make sure you understand the problem
      a. Identify the inputs
      b. Identify what the output should look like
   B. If it's a large scale problem, identify possible smaller components

2. **Pattern Recognition:** Observing patterns, trends, and regularities in data
   A. Inspect your data
      a. Make sure the inputs and outputs you've identified make sense
      b. Look for trends or possible fringe cases

3. **Abstraction:** Identifying the general principles that generate these patterns
   A. Work out a problem by hand
   B. As you work out by hand, attempt to think beyond a single example

4. **Algorithm Design:** Developing the step by step instructions for solving the problem

A. Begin to formalize your steps into pseudocode
B. Identify available software tools
    a. These may be available packages, things you've written in the past, or built-in python functionality
C. Implement your pseudocode

1.8.1. Guided Exercise: Apply Computational Thinking

Given the paragraph below, identify which sentence has the most words in it.

```
'''Security has been tightened in Kenya ahead of a controversial re-run of the p
residential election which is being boycotted by the main opposition. Kenya's Pr
esident Uhuru Kenyatta, who is seeking a second term, has urged people to vote a
nd remain peaceful. Opposition leader Raila Odinga, who has pulled out of the el
ection re-run, has called on his supporters to boycott it. Mr. Kenyatta was anno
unced the winner in an 8 August vote but the poll is being held again because of
"irregularities".'''
```

Instructor Guidance: Work this problem out with the students. Highlight that the end result should be to have a process that someone who doesn't understand the goal of what they're doing can follow and consistently get the right result.

Instructor Guidance: In step one below, emphasize that we should be thinking about how our code will work on different paragraph inputs. This can also be emphasized in step three (abstraction). Further explain the use of python for approaching this problem by asking how the approach would be different if the string was a whole article, or whole newspaper.

Step 1. Decomposition

The first step to tackling any problem is to make sure you understand it, and break it down into smaller, more manageable parts. All computational problems have some input and some output. Our solution is going to start with our input and produce the output we want.

In this example problem, our input is a paragraph. In this case, an example paragraph is given to us, but the great thing about coding is that even if the input changes (i.e., if we're given a *new* paragraph), re-applying an algorithm is easy. What is the output we want? We've been asked to identify the longest sentence in a paragraph. The output will be the sentence with the most words.

This problem has one major task, but we can break it up into smaller tasks to make algorithm design easier. After we accept the input, we need a way to split it up into individual sentences, count the words in each sentence, and determine which one has the most.

Step 2. Pattern Recognition

Looking at our example paragraph, what can we use to divide it into sentences? What can we use to divide each sentence into words so we can count them?

Here we should start thinking about *fringe cases*. While most English sentences end with a period, sometimes they also end with an exclamation point ("!") or a question mark("?"). On the other hand, a period does not always denote the end of a sentence. For example, it could be part of a decimal number (2.15) or an abbreviation, such as "Mr." As you can see, fringe cases appear when the first approach we think of is too broad or too narrow.

Fringe Cases: These appear when our approach is either...

- Too Broad: Our rule captures cases we don't want it to capture
- Too Narrow: Our rule fails to capture cases we want it to capture

Step 3. Abstraction

We've been given an example to work through. But we want our solution to work for other similar inputs. That means our solution shouldn't depend on features of the example paragraph that might not appear in other valid paragraphs. We can use any patterns and fringe cases we identified in the last step to help us.

Step 4. Algorithm Design

Designing an algorithm starts with writing something called *pseudocode*. Pseudocode is a detailed yet human-readable description of what a computer program or an algorithm must do. It is written in natural language instead of a programming language, but nonetheless captures the same logical steps. In this case, our pseudocode might look something like this.

Pseudocode:

1. Split the paragraph by periods to separate sentences
2. For each sentence:
    A. Count how many words are in the sentence
    B. Compare the word count with the longest sentence you've seen so far
    C. If it's the longest sentence, hold on to it, otherwise move to the next sentence
3. Finally, print out the longest sentence and its word count

Run through the pseudocode by hand with your example, and make sure it works like you'd expect. We haven't yet covered the the tools available to you in Python, but if you know of any tools that will be helpful, your pseudocode is a good place to note that. We haven't taught you how to code yet, but here's what our algorithm would look like in Python.

In [ ]:

```
paragraph = """Security has been tightened in Kenya ahead of a controversial re-run of th
election which is being boycotted by the main opposition. Kenya's President Uhuru Kenyatt
second term, has urged people to vote and remain peaceful. Opposition leader Raila Odinga
of the election re-run, has called on his supporters to boycott it. Mr. Kenyatta was anno
an 8 August vote but the poll is being held again because of "irregularities"."""

## 1. Split the paragraph on "."
sentences = paragraph.split(".")

longest_sentence = ""

## 2. For each sentence...
for sentence in sentences:

    ## 2A-B. Count words and compare to the longest sentence we've seen so far
    if len(sentence.split(" ")) > len(longest_sentence.split(" ")):

        ## 2C. If it's the longest, hold on to it
        longest_sentence = sentence + "." #

## 3. Finally, print the longest sentence and its word count
print(len(longest_sentence.split(" ")))
print(longest_sentence)
```

1.9. Practice Python

Let's get some practice with Python basics before we end the lesson. But first, consider this foundational concept of coding in general.

1.9.1. Hard-Coding vs. Programmatic Coding

Hard-coding is when you type explicit values in your code. *Programmatic coding*, on the other hand, is using variables wherever possible. By writing programmatic code, you make your scripts useful to others as well as yourself. For example, let's say we write a script that approximates the cost of hosting a conference for a given number of people, with set expenses per person. With 50 attendees, our code could look something like this:

In [ ]:

```
total_conf_cost = 50 * 8.00 + 50 * 1.50 + 50 * 14.25 + 50 * 0.75 + 50 * 10.00

print('Number of attendees:', 50)
print('Cost of conference:', total_conf_cost)
```

Now, we received a late addition to our list of attendees. This requires that we adjust the code in at least six separate locations, and remember what each number's significance is in real life.

What would happen if the price of one component of the conference changed? What if the calculations were more complex? What if another number in the calculation was the same as the number of attendees?

Let's try pulling out the number of attendees and storing it as a variable.

In [ ]:

```
num_attendees = 51

cost_box_lunch = num_attendees * 8.00
cost_drinks = num_attendees * 1.50
cost_tshirt = num_attendees * 14.25
cost_icecream = num_attendees * 0.75
cost_goody_bags = num_attendees * 10.00

total_conf_cost = cost_box_lunch + cost_drinks + cost_tshirt + cost_icecream + cost_goody
print('Number of attendees:', num_attendees)
print('Cost of conference:', total_conf_cost)
```

Imagine this is part of a larger, more complicated script in which you use the number of attendees two dozen times in various ways. Coding programmatically by storing the value to a variable allows us to make changes much faster and makes our code much less error-prone in the long run. Now our code is a little more robust thanks to our smart use of variables. But what if we wanted to find the cost of hosting a conference for a group of 500? If we're hard-coding, we'd have to go in a swap in `500` everywhere we see the value `51` in our code above. This is doable with only a few changes here, but what if our script used that attendee value in a hundred other places? Do you trust yourself to remember to swap every single value in your code? Better question: Why even bother if you can use a variable to do it for you?

1.9.2. Storing Values In Variables

Exercise: Store two numbers inside two different variables. Subtract one number from the other using the minus sign (`-`). Multiply them together using Python's multiplication operator (`*`).

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
n1 = 56
n2 = 14.85

print(n1 - n2)
print(n1 * n2)
```

Exercise: In Python, a text-based data point is called a string. Strings in Python are wrapped in quotation marks (`""`). Store a string in a variable, then use the `print()` function to print it out.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
text = 'Some text is here printed'
print(text)
```

Exercise: The formula for finding the area of a circle with a given radius is below, but the cell below will throw an error because `r` is not defined. Create a variable `r` that stores a radius of your choice then use the code below to find the area of a circle with that radius. After you get it to work, change the value of your radius variable and run it again. You should see a different output.

In [ ]:

```
## YOUR CODE GOES HERE ##
PI = 3.14

area = PI * r ** 2

area
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
r = 30.654321
PI = 3.14

area = PI * r ** 2

area
```

Exercise: The `len()` function in Python finds the length of something. When called on some text, it will return the number of characters the text is made up of. Use it by putting some text or a variable that holds some text into the parentheses, like so:

```
len(some_text)
```

Create a variable that holds some text. Then use the `len()` function to find the length of it. After getting that to work, change the text that your variable holds and run the code again.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
text = 'Four score and seven years ago...'

len(text)
```

Exercise: Python can capture the concept of true and false. In coding, these are called Boolean values, and in Python they are capitalized like `True` and `False`. Store a Boolean value in a variable and print it using `print()`.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
valid_id = True
print(valid_id)

is_raining = False
print(is_raining)
```

Exercise: Python scripts can also store multiple data values in a single object called a list. You can make a list by wrapping any number of data values in a set of square brackets. Each value must be separated by a comma, like so:

```
my_list = [1, 2, 3, 4, 5, 6]
```

Create your own list of values and store it to a variable.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
my_list = ['one', 'two', 'three', 'four', 'five']
```

Exercise: The `len()` function works on lists too. Call the `len()` function on the list you just created.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
len(my_list)
```

1.10. Appendix

Jupyter Notebook Shortcuts

Shortcut	Description
Shift + Enter	Execute the contents of the cell, and move to the next cell
Ctrl + Enter	Execute the contents of the cell, but stay with this cell
Alt + Enter	Execute the contents of the cell, insert a new cell below
Enter	enter Edit Mode
Esc	enter Command Mode
m	Convert a cell to Markdown
y	Convert a cell to Code
l	Display line numbers in the selected cell
a	Insert a cell above the selected cell
b	Insert a cell below the selected cell
c	Copy the selected cell
x	Cut the selected cell
v	Paste cell below the selected cell
dd	Delete selected cell
z	Undo a cell deletion
Up Arrow	Highlight the cell above
Down Arrow	Highlight the cell below