Topic 1 - History of Computer Science and Programming Basics

- Data Structures
- Iteration and Recursion
- Organize and Modularize Systems using object classes and methods
- Algorithms
- Complexity of Algorithms

What is Computer Science?

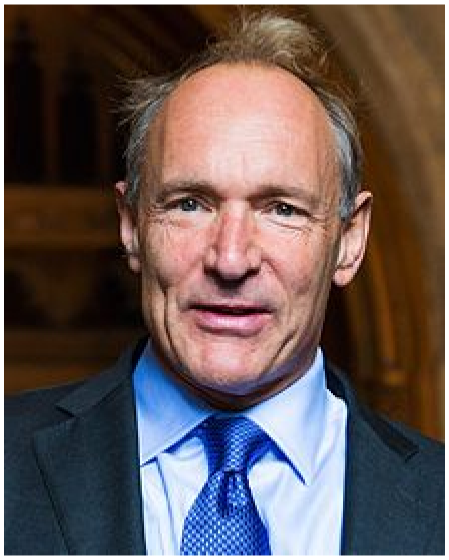
Computer Science is the study of processes that interact with data and that can be represented as data in the form of programs. It enables the use of algorithms to manipulate, store, and communicate digital information.

Historical Figures

Alan Turing was British mathematician, computer scientist, and cryptanalyst. He is credited with desiging and creating "The Bombe", a electro-mechanical device used to help decipher the German Enigma-machine-encrypted secret messages during WWII.

- He has been called one of the fathers of modern computer science

Margaret Hamilton is an American computer scientist, systems engineer, and business owner. She coined the term "software engineering", as very little people had the respect for that field than she felt there should be. She was the director of the Software Engineering Division at MIT Instrumentation Laboratory, which developed on-board flight software for the Apollo space program



Sir Tim Berners-Lee is an English computer scientist, best known as the inventor of the World Wide Web.

What is Computation?

Computation is any type of calculation that includes both **arithmetical** and **non-arithmetical** steps, and follows a well-defined model (i.e. an algorithm).

What is Computational Thinking?

A set of problem-solving methods that involve expressing problems and their solutions in ways that a computer could execute

What does a Computer DO?

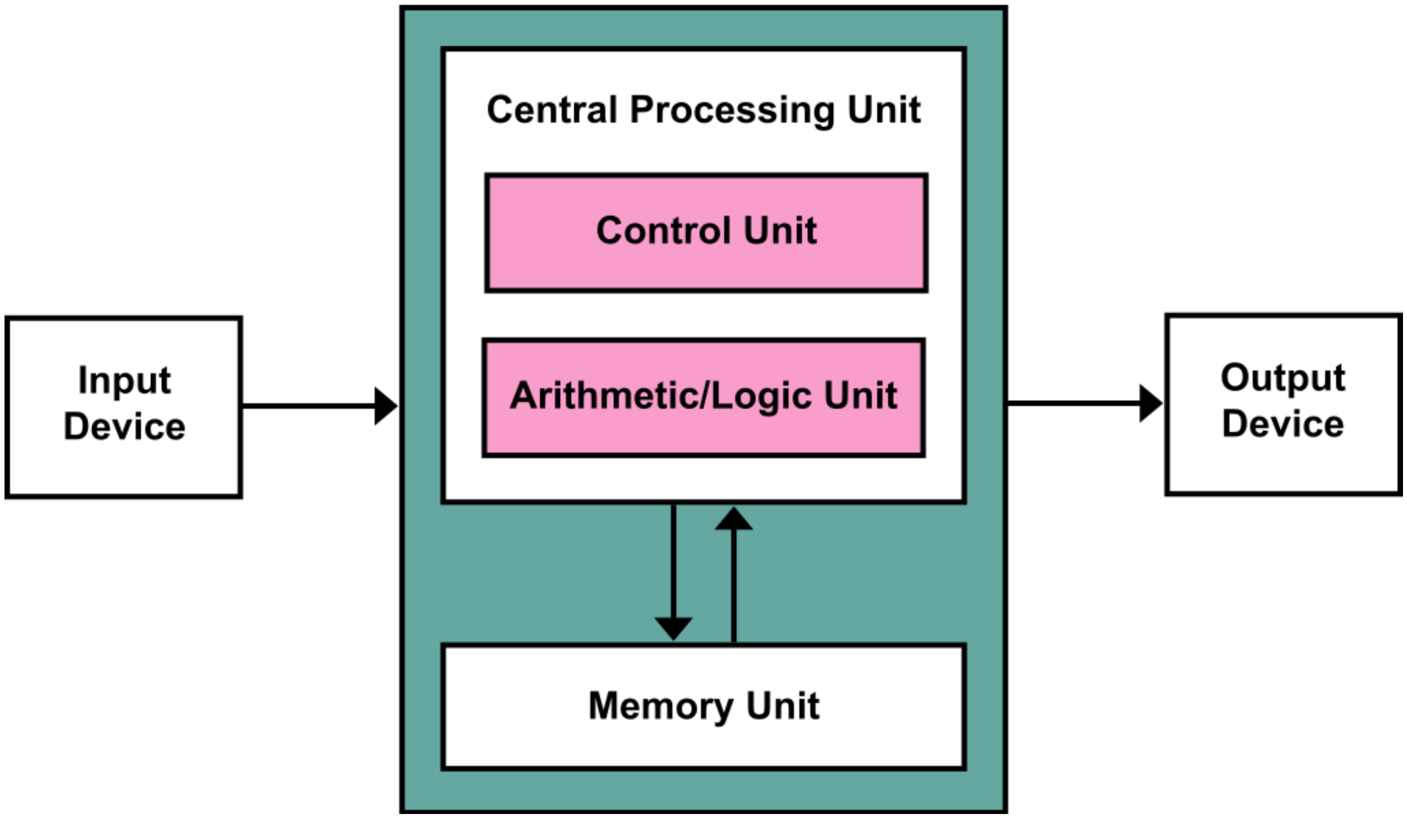
- Fundamentally it:
  - Performs calculations
    - A billion per second
  - Remembers results
    - terabytes of storage!
  - What kinds of calculations?
    - Built-in to the language
    - Ones that your define as the programmer
  - Computers do what you tell them
    - Say it with me, computers are dumb, you are smart

Who created our "modern" computer architecture?

John von Neumann

von Neumann was a Hungarian-American mathematician, physicist, computer scientist, and engineer. He created what is called the von Neumann architecture for computer systems.

Von Neumann Architecture



What Makes Up A Computer

A computer is made up of many different physical parts, called **hardware**. Hardware is more expensive to upgrade and more time consuming than software.

Computer Hardware

Basic parts of a Computer

Stored Program Computer

Sequence of instructions stored inside computer

- Built from predefined set of primitive instructions
  - Arithmetic and Logic
  - Simple Tests
  - Moving Data

Special program, the interpreter, executes each instruction in order

- Use tests to change flow of control through sequence
- Stops when done

Types of Knowledge

- **Declarative** - Statements of facts (the sky is blue, water is wet, etc)
- **Imperative** - A recipe, or "how-to"

Algorithms are like baking cookies

Exactly! An algorithm is a fancy word for describing a sequence of simple steps. You've been following algorithms all your life! Down below, describe the steps you need to follow in order to brush your teeth. Break it down to the smallest movement. Minimum 10 steps.

An algorithm is a flow of control process that determines when each step is executed, and provides a means of determining when to stop.

What is a program?

A program is a sequence of definitions and commands

- Definitions evaluated
- Commands executed by the Python interpreter in a shell

Commands (statements) instruct the interpreter to do something

Can be typed directly in a shell, or stored in a file that is read into the shell and evaluated.

We've talked a lot. Let's code for a bit

Follow along with me at the front and we're going to go through some of the most basic functions in Python.

In [1]:
```
# the print function

print("hello world")  # run this cell and see the output
```

```
hello world
```

The above is the traditional first steps to becoming a computer scientist. print displays the data onto the console, or the computer screen.

When you want your program to output so that the user (you or any one else) can see it, you must use the print function

Basic Primitives

A low-level object or operation from which higher level, more complex objects and operations can be constructed.

- Alan Turing showed that you can compute anything using 6 primitives, however, modern programming languages have more convenient set of primitives.
- You can abstract methods to create new primitives.
- Anything computable in one language is computable in any other programming language

What?

Primitives in English: words

You use the alphabet to construct words, and you put words together to create sentences. You use sentences to create paragraphs. As you put words together, you attach and create meaning towards them.

Primitives in programming languages: numbers, strings, simple operators

In [1]:

```
# Variables

x = 3 # the computer sees 3 when x is present
y = x

print("The value of x is " + str(x))
print("The value of y is " + str(y))
print("The value of x times y is " + str(x*y))
```

```
The value of x is 3
The value of y is 3
The value of x times y is 9
```

Variables are one of the most common and useful parts of any programming language. They enable you to store data and information in a particular spot in the computer's memory.

Think of a variable like the cubby you were given back in kindergarten. You were able to put your stuff in there, and it would be safe. But that cubby was used before you were there, and it can be used after you are finished kindergarten.

In [1]:
```
# Strings

greeting = "hello world"
print(greeting) #displays to the console remember
```

```
hello world
```

The above variable greeting has text within quotations(" "). That says to the computer that it is the String datatype.

A string contains characters that represent language to us. The computer doesn't understand them as we do however, just little bits of data attached to characters.

A string doesn't have to be text, you can use #'s in a string, but you cannot do any arithmetic (math) with strings.

Numbers - Different Types

In [ ]:
```
# Integers and floats

int x = 3
float y = 3.0

print(x)
print(y)

# What is the difference between those two?
```

Integers and floats are two of the most basic datatypes.

Integers are whole numbers, floats are real numbers (they have decimals).

Creating Recipes

A programming language provides a set of primitive operations:

- Expressions are complex, but legal combinations of primitives in a programming language
- Expressions and computations have values and meanings in a programming language

Aspects of Languages

Syntax -- All languages have sets of rules on how characters and words are supposed to interact with each other. Programming languages are no different.

English: "cat dog boy" --> makes zero sense, not syntactically valid.

"cat hugs boy" is syntactically valid

In [2]:
```
print("hi"5) # not syntactically valid, outputs an error.
```

```
  File "<ipython-input-2-41c88f305587>", line 1
    print("hi"5) # not syntactically valid, outputs an error.
              ^
SyntaxError: invalid syntax
```

In [3]:
```
print(3.2 * 5) # syntactically valid
```

```
16.0
```

Think about grammar with English and spoken languages. There are different types of grammatical mistakes, some make it impossible to read and understand, others affect the readability and as those pile up, it becomes hard to understand.

Eg) "I are hungry" is a syntactically valid sentence, but is not grammatically correct.

This is called a static semantic error <-- important

We want to find the percentage of people who like chocolate

In [ ]:
```
chocolate = 20
people = 150

print(chocolate / people)

# Where is the static semantic error?
```

Have you ever had an argument, and someone says that you are just arguing semantics? Semantics means that you are arguing about different interpretations rather than the core meaning.

In programming, semantics is the meaning associated with a syntactically correct string of symbols with no static semantic errors.

English: "Flying planes can be dangerous"

The above statement has multiple meanings, try and count all the different ways it could be understood.

Programming Languages: The actual code only has one meaning, however, it might not be what a programmer intended.

Where things go wrong

Syntactic Errors

- Common, easily caught

Static Semantic Errors

- Some languages check for these before running program (ROBOTC)
- Can cause unpredictable behaviour

No Semantic errors, but different meaning than what the programmer intended

- Program crashes, stops running
- Program runs forever (infinite loop)
- Program gives an answer but different than expected

Objects

Programs manipulate data objects

Objects have a type that defines the kinds of things programs can do to them

- Mr. Wain is a human so he can walk, speak English etc

Objects can be two types:

- Scalar - cannot be subdivided
- Non-scalar - have an internal structure that can be accessed

Scalar Objects

- int - represents integers (0, 1, 2, 3, 4, 5 ...)
- float - represents real numbers (3.14159, 1.0, ...)
- bool - represents Boolean values (True or False)
- NoneType - special and has one value, None

Can use type() to see the type of an object

Type Conversions (CAST)

Can convert object of one type to another

```
In [1]: float(3) #converts integer 3 to float 3.0

        int(3.9) #converts float 3.9 to integer 3

Out[1]: 3
```

Printing to Console

To show output from code to a user, use print command

```
In [ ]: x = 3 + 2
        print(x)

        y = 3 + x
        print(y)
```

Let's work with strings and code for a bit

```
In [1]: Greeting = "Hello World!"
        print(Greeting)

        Letter = Greeting[2]
        print(Letter)

        Hello World!
        l
```

```
In [2]: # The Len Function -- use to find out the length of a string

        name = "Dave"
        len(name)

Out[2]: 4
```

```
In [3]: # Converting data types

        x = 23 # an integer

        #str(value) # will convert a given value to a string
        str(x) # is now a string '23'

Out[3]: '23'
```

```
In [4]: #Example 1 - age as an ordinary integer
        age = 15
        print(age - 2)

        13
```

```
In [5]: #Example 2 - age converted to a string
        age = 15
        print(str(age) - 2) #this will result in an error, why?

        ---------------------------------------------------------------------------
        TypeError                                 Traceback (most recent call last)
        <ipython-input-5-83537488e005> in <module>
              1 #Example 2 - age converted to a string
              2 age = 15
        ----> 3 print(str(age) - 2) #this will result in an error, why?

        TypeError: unsupported operand type(s) for -: 'str' and 'int'
```

```
In [6]: #Example 3 - Concatenate - adding strings and integers together

        age = 15
        Statement = "You are "
        print(Statement + str(age))

        #Why is there an error with this?

        You are 15
```

```
In [7]: age = 15
        statement = "You are "

        print(statement + str(age))

        You are 15
```

Operators in Programming Languages

If you thought you have escaped math, you are sorely mistaken. This field and class is filled with math.

Operators on ints and floats:

1. i + j --> the sum
2. i - j --> the difference
3. i * j --> the product
4. i / j --> division --> Result is always a float

1, 2, and 3:

- If both are int, the result is int
- If either or both are floats, the result is float

Mathematical Operators in Programming Languages

1. ** = exponent
2. • = multiplication
3. / = division
4. +/- = addition and subtraction
5. % = moduluo (finds the remainder after division)

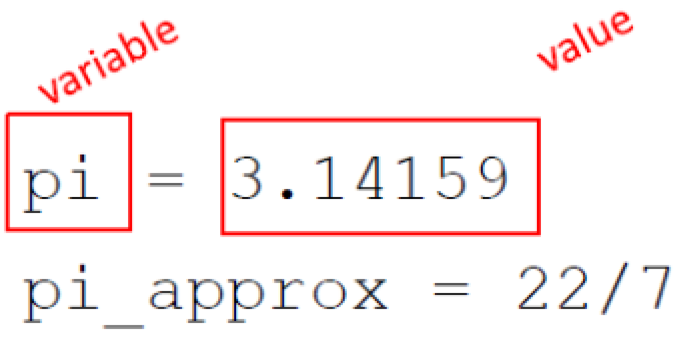
Order of Operations

Parentheses are used to tell Python to do these operations first

Operation precedence without parentheses:

1. **
2. *
3. /
4. +/- executed left to right, as appear in expression

Binding Variables and Values



Equals sign is an assignment of a value to a variable name

Value is stored in computer memory

An assignment binds name to value

Retrieve value associated with name or variable by invoking the name, by typing pi

```
In [8]: pi = 3.14159 # variable name = value
        pi_approx = 22/7 # computer will run the expression and bind the result to the variable name

        print(pi)
        print(pi_approx)
```

```
3.14159
3.142857142857143
```

Abstracting Expressions

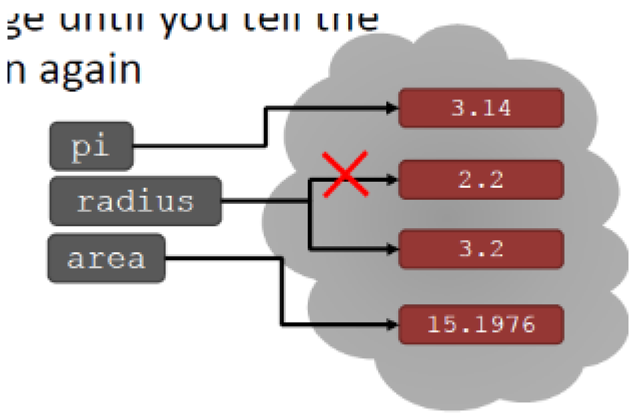
Why give names to values of expressions?

- To reuse names instead of values
- Easier to change code later

In [9]:
```
pi = 3.14159
radius = 2.2

area = pi * (radius**2)

print(area)
```

```
15.205295600000001
```

Changing Bindings



You can **re-bind** variable names using new assignment statements

- Previous value may still be stored in memory, but lost in the handle for it
- Value for area does not change until you tell the computer to do the calculation again

In [1]:
```
pi = 3.14159
radius = 2.2

area = pi*(radius**2)
print(area)

radius = radius + 1
print(area)

area = pi*(radius**2)
print(area)
```

```
15.205295600000001
15.205295600000001
32.169881600000004
```

Prompting User Inputs

We can use **input()** to grab some data from our end user.

In [ ]:
```
age = input("What is your age? ")
print(age) # note, this will be a string. Anything taken in as an input is a s
tring unless converted to a different data type
```

In [ ]:
```
# We are going to change the input to an int

age = int(input("What is your age? "))
print(age) # this is now an integer
```

TASK #1 - Prompting User for some data

1. Prompt the user for 5 numbers
2. Each number should be assigned an appropriate variable name (eg. num1, num2, etc)
3. After 5 numbers have been entered, the sum of the numbers should be outputted as:
   - "The sum of your 5 numbers is x."
4. The final line should show the average of the 5 numbers, and should read like this:
   - "The average of your 5 numbers is x."

Think carefully about which data types to use in the sum and average calculations!!!

Topic 2 - Branching, Iteration

- String Object Type
- Branching and Conditionals
- Indentation
- Iteration and Loops

Take out the History of Computers worksheet and follow along with the video below. **This is FOR MARKS.**

In [2]:
```
from IPython.display import HTML

HTML('<iframe width="560" height="315" src="https://www.youtube.com/embed/3xdm
EwTIsd0" frameborder="0" allow="accelerometer; autoplay; encrypted-media; gyro
scope; picture-in-picture" allowfullscreen></iframe>')
```

```
/opt/conda/lib/python3.7/site-packages/IPython/core/display.py:701: UserWarni
ng: Consider using IPython.display.IFrame instead
  warnings.warn("Consider using IPython.display.IFrame instead")
```

Out[2]:

Strings

Letters, Special Characters, Spaces, Digits

Enclosed in quotation marks or single quotes

In [ ]:
```
hi = "hello there"
```

Concatenate strings

In [1]:
```
hi = "hello there"
name = "ana"
greet = hi + name
print(greet)

greeting = hi + " " + name
print(greeting)
```

```
hello thereana
hello there ana
```

Do some operations on a string as defined in Python docs

```
In [3]: silly = hi + " " + name * 3
        print(silly)
```

```
hello there anaanaana
```

Input / Output: print

Used to output stuff to console

Keyword is print

```
In [ ]: x = 1
        print(x)

        x_str = str(x)

        print("My favourite number is ", x, ". ", "x = ", x)
        print("My favourite number is ", x_str, ". ", "x = ", x_str)
```

Comparison Operators on int, float, and string

i and j are variable names

Comparisons below evaluate a Boolean

- i > j
- i >= j
- i < j
- i <= j
- i == j --> EQUALITY TEST, TRUE if i is the same as j
  - 1 does not equal 1.0
- i != j --> INEQUALITY TEST, TRUE if i is not the same as j

Logic Operators on bools

a and b are variable names (with Boolean values)

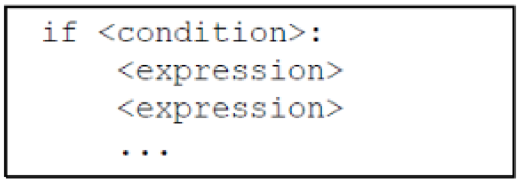
- not a --> TRUE if a is False. FALSE if a is True
- a and b --> TRUE if both are True
- a or b --> TRUE if either or both are True

In [4]:
```python
# Comparison Example

pset_time = 15
sleep_time = 8
print(sleep_time > pset_time) # what will print here?

derive = True
drink = False

both = drink and derive
print(both) # what will print here?
```

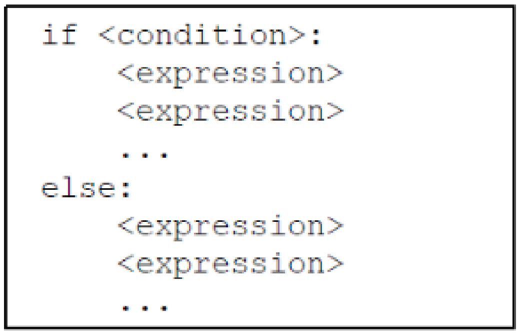
```
False
True
```

Control Flow - Branching

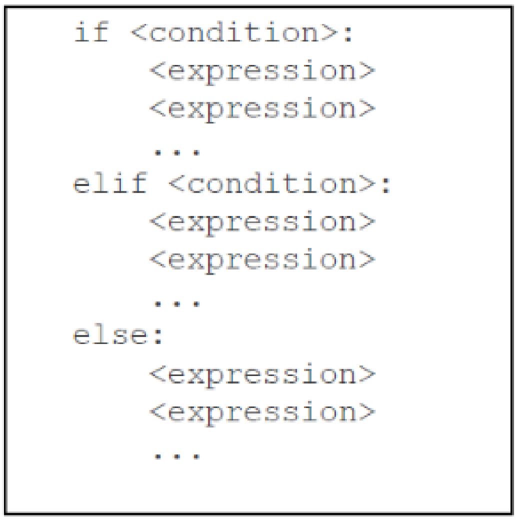
If/Else Statements are key here.



```
if <condition>:
    <expression>
    <expression>
    ...
```



```
if <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```



```
if <condition>:
    <expression>
    <expression>
    ...
elif <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```

< condition > has a value True or False

Evaluate expressions in that block if < condition > is True

Indentation

Is incredibly important in Python

Indentation is how you denote blocks of code

In [ ]:
```python
x = float(input("Enter a number for x: "))
y = float(input("Enter a number for y: "))

if x == y:
    print("x and y are equal.")
    if y != 0:
        print("Therefore, x / y is ", x/y)
elif x < y:
    print("x is smaller. ")
else:
    print("y is smaller. ")

print("Thanks! ")
```

- Legend of Zelda – Lost Woods
- keep going right, takes you back to this same screen, stuck in a loop

```
if <exit right>:
    <set background to woods_background>
    if <exit right>:
        <set background to woods_background>
        if <exit right>:
            <set background to woods_background>
            and so on and on and on...
        else:
            <set background to exit_background>
    else:
        <set background to exit_background>
else:
    <set background to exit_background>
```

- Legend of Zelda – Lost Woods
- keep going right, takes you back to this same screen, stuck in a loop

```
while <exit right>:
    <set background to woods_background>
<set background to exit_background>
```

Control Flow: while loops

```
while <condition>:
    <expression>
    <expression>
    ...
```

< condition > evaluates to a Boolean

If < condition > is True, do all the steps inside the code block

Check < condition > again

Repeat until < condition > is False

In [ ]:
```
# While Loop Example

n = input("You're in the Lost Forest. Go left or right? ")

while n == "right":
    n = input("You're in the Lost Forest. Go left or right? ")
print("You got out of the Lost Forest! ")
```

Control Flow: while and for loops

Iterate through numbers in a sequence

In [ ]:
```
# this is more complicated with a while loop than a for loop

n = 0

while n < 5:
    print(n)
    n = n + 1
    # n += 1 - same as above
```

In [ ]:
```
# shortcut with for loop

for n in range(32361):
    print(n)
```

For loops are useful when you know how many times you want the code to loop for.

While loops are more useful when you do not know the exact number of times you want it to run.

Control Flow: for loops

```
for <variable> in range(<some_num>):
    <expression>
    <expression>
    ...
```

- Each time through the loop, < variable > takes a value
- First time, < variable > starts at the smallest value
- Next time, < variable > gets previous value + 1

Range (start, stop, step)

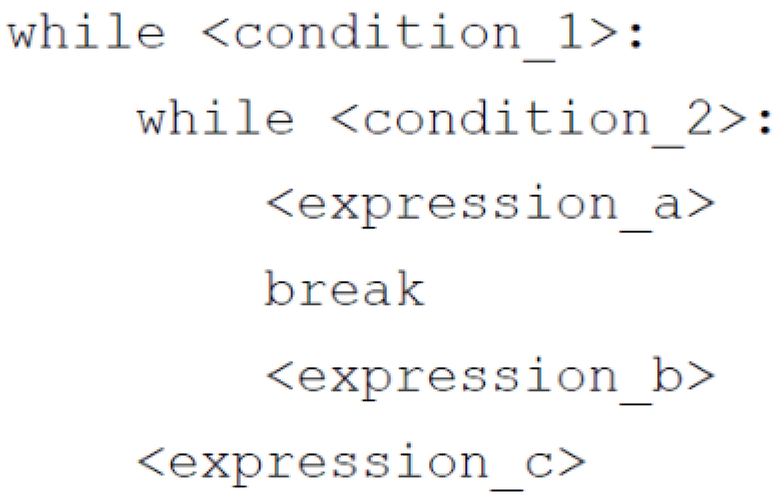
Default values are start = 0 and step = 1 and optional

Loop until value is stop -1

In [ ]:
```
mysum = 0

for i in range(7, 10):
    mysum += i # i = 7 + 8 + 9 = 24
print(mysum)
```

In [ ]:
```
mysum = 0

for i in range(5, 11, 2):
    mysum += i
print(mysum)

# Guess 1: 15
# Guess 2: 90
# Guess 3: 21 = 5 + 7 + 9
```

Break Statement


```
while <condition_1>:
    while <condition_2>:
        <expression_a>
        break
        <expression_b>
    <expression_c>
```

Immediately exits whatever loop it is in

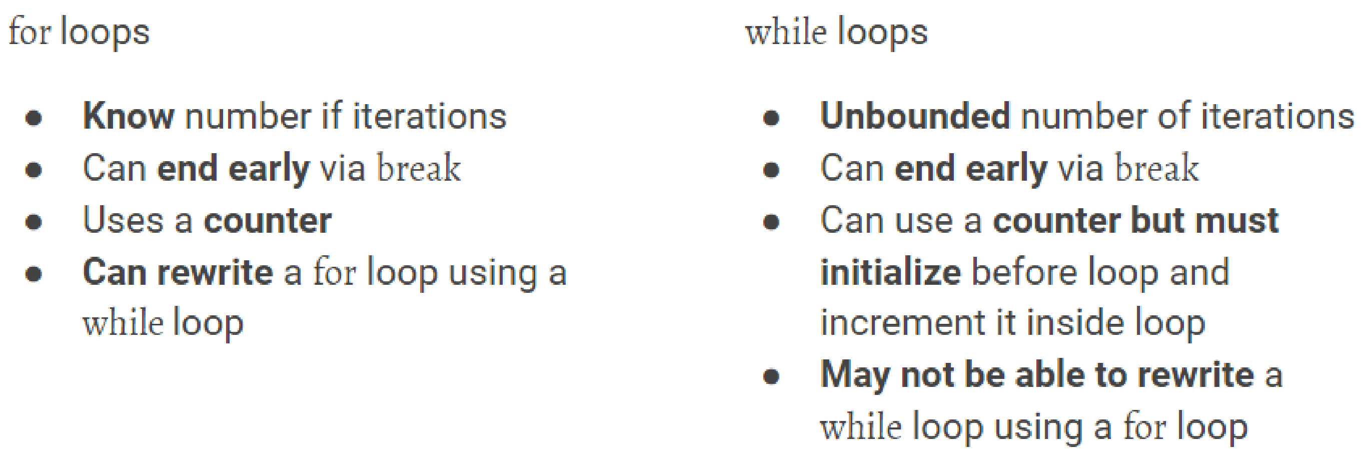
Skips remaining expressions in code block

Exits only innermost loop!

In [ ]:

```
mysum = 0

for i in range (5, 11, 2):
    mysum += i
    if mysum == 5:
        break
        mysum += 1

print(mysum)

# What will print?
```

For versus While Loops



for loops

- **Know** number if iterations
- Can **end early** via break
- Uses a **counter**
- **Can rewrite** a for loop using a while loop

while loops

- **Unbounded** number of iterations
- Can **end early** via break
- Can use a **counter but must initialize** before loop and increment it inside loop
- **May not be able to rewrite** a while loop using a for loop