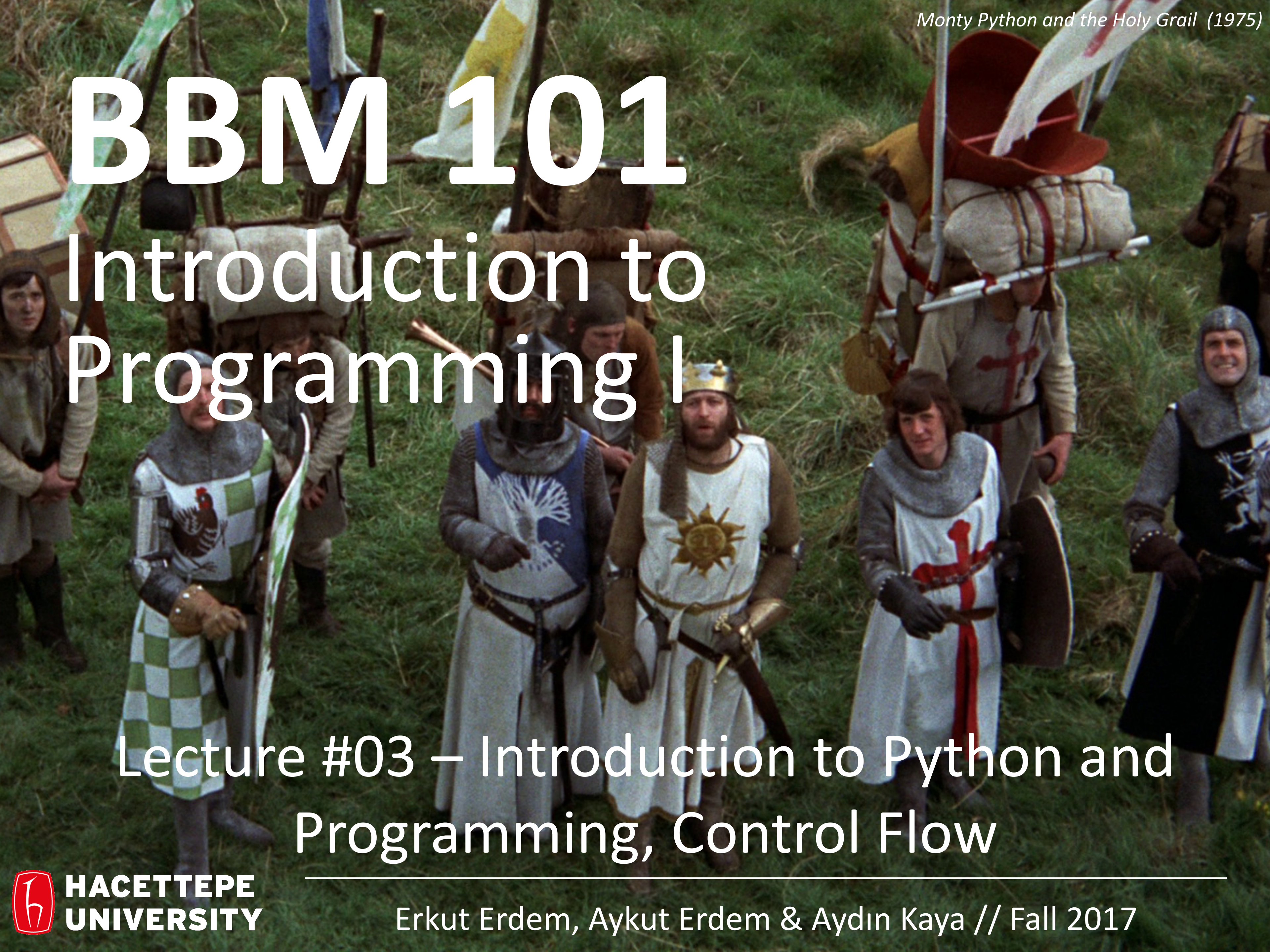

Monty Python and the Holy Grail (1975)

BBM 101

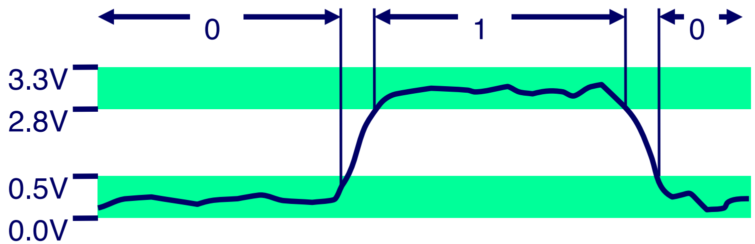
Introduction to Programming I

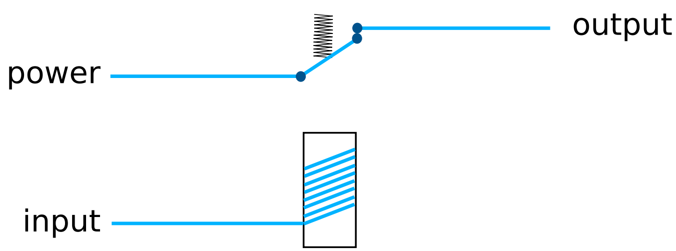
Lecture #03 – Introduction to Python and Programming, Control Flow

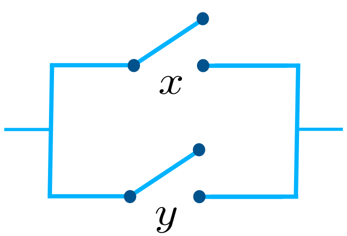
HACETTEPE UNIVERSITY

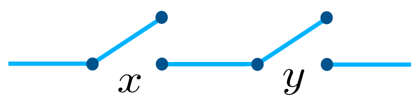
Erkut Erdem, Aykut Erdem & Aydın Kaya // Fall 2017

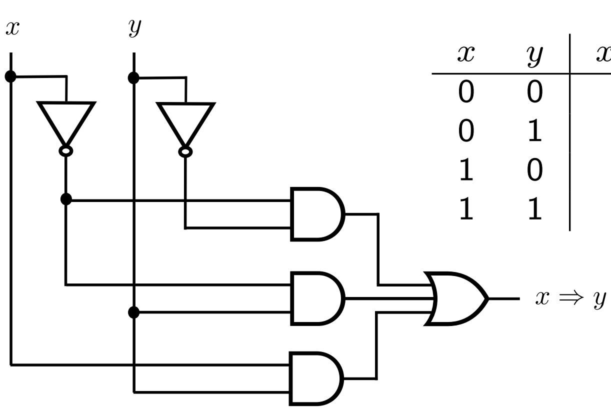
Last time... **How to build computers**



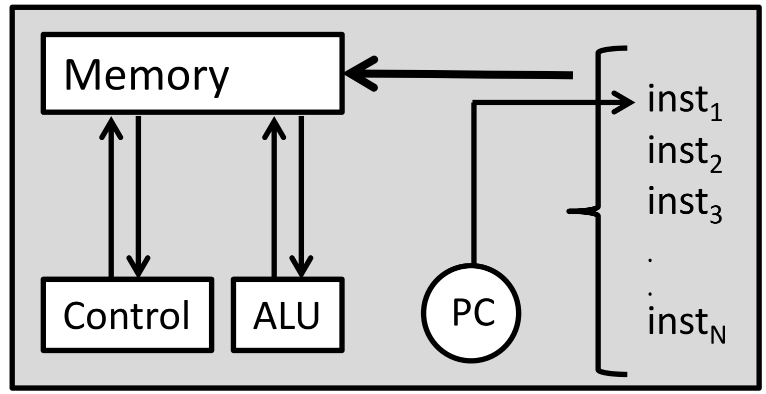








x	y	$x \Longrightarrow y$
0	0	1
0	1	1
1	0	0
1	1	1



The Harvey Mudd Miniature Machine (HMMM)

triangle1.hmmm: Calculate the approximate area of a triangle.

```
0    read    r1        # Get base b
1    read    r2        # Get height h
2    mul     r1 r1 r2  # b times h into r1
3    setn    r2 2
4    div     r1 r1 r2  # Divide by 2
5    write   r1
6    halt
```

```
$ python hmmmAssembler.py -f triangle1.hmmm -o triangle1.b

----------------------
| ASSEMBLY SUCCESSFUL |
----------------------

0 : 0000 0001 0000 0001        0    read    r1        # Get base b
1 : 0000 0010 0000 0001        1    read    r2        # Get height h
2 : 1000 0001 0001 0010        2    mul     r1 r1 r2  # b times h into r1
3 : 0001 0010 0000 0010        3    setn    r2 2
4 : 1001 0001 0001 0010        4    div     r1 r1 r2  # Divide by 2
5 : 0000 0001 0000 0010        5    write   r1
6 : 0000 0000 0000 0000        6    halt
```

```
$ python hmmmSimulator.py -f triangle1.b -n
4
5
10
```

Lecture Overview

- Programming languages (PLs)

- Introduction to Python and Programming

- Control Flow

Disclaimer: Much of the material and slides for this lecture were borrowed from
—E. Grimson, J. Guttag and C. Terman MIT 6.0001 class
—Ruth Anderson, Michael Ernst and Bill Howe's CSE 140 class
—Swami Iyer's Umass Boston CS110 class

Lecture Overview

- Programming languages (PLs)
- Introduction to Python and Programming
- Control Flow

Programming Languages

- Syntax and semantics
- Dimensions of a PL
- Programming paradigms

Programming Languages

- An artificial language designed to express computations that can be performed by a machine, particularly a computer.

- Can be used to create programs that control the behavior of a machine, to express algorithms precisely, or as a mode of human communication.

- **e.g.**, C, C++, Java, Python, Prolog, Haskell, Scala, etc..

Creating Computer Programs

- Each programming language provides a set of primitive operations.
- Each programming language provides mechanisms for combining primitives to form more complex, but legal, expressions.
- Each programming language provides mechanisms for deducing meanings or values associated with computations or expressions.

Aspects of Languages

- Primitive constructs
  - Programming language – numbers, strings, simple operators
  - English – words

- Syntax – which strings of characters and symbols are well-formed
  - Programming language –we'll get to specifics shortly, but for example 3.2 + 3.2 is a valid C expression
  - English – "cat dog boy" is not syntactically valid, as not in form of acceptable sentence

Aspects of Languages

- Static semantics – which syntactically valid strings have a meaning

  – English – “I are big” has form <noun> <intransitive verb> <noun>, so syntactically valid, but is not valid English because “I” is singular, “are” is plural

  – Programming language – for example, <literal> <operator> <literal> is a valid syntactic form, but 2.3/’abc’ is a static semantic error

Aspects of Languages

- Semantics – what is the meaning associated with a syntactically correct string of symbols with no static semantic errors
  - English – can be ambiguous
    - "They saw the man with the telescope."
  - Programming languages – always has exactly one meaning
    - But meaning (or value) may not be what programmer intended

Where Can Things Go Wrong?

- Syntactic errors
  - Common but easily caught by computer
- Static semantic errors
  - Some languages check carefully before running, others check while interpreting the program
  - If not caught, behavior of program is unpredictable
- Programs don’t have semantic errors, but meaning may not be what was intended
  - Crashes (stops running)
  - Runs forever
  - Produces an answer, but not programmer’s intent

Our Goal

- Learn the syntax and semantics of a programming language
- Learn how to use those elements to translate "recipes" for solving a problem into a form that the computer can use to do the work for us
- Computational modes of thought enable us to use a suite of methods to solve problems

Dimensions of a Programming Language
Low-level vs. High-level

- Distinction according to the level of abstraction

- In low-level programming languages (e.g. Assembly), the set of instructions used in computations are very simple (nearly at machine level)

- A high-level programming language (e.g. Python, C, Java) has a much richer and more complex set of primitives.

Dimensions of a Programming Language
General vs. Targeted

- Distinction according to the range of applications

- In a general programming language, the set of primitives support a broad range of applications.

- A targeted programming language aims at a very specific set of applications.
  - **e.g.**, MATLAB (matrix laboratory) is a programming language specifically designed for numerical computing (matrix and vector operations)

Dimensions of a Programming Language

Interpreted vs. Compiled

- Distinction according to how the source code is executed

- In interpreted languages (e.g. LISP), the source code is executed directly at runtime (by the interpreter).
  - Interpreter control the flow of the program by going through each one of the instructions.

- In compiled languages (e.g. C), the source code first needs to be translated into an object code (by the compiler) before the execution.

Programming Language Paradigms

- **Functional**
  - Treats computation as the evaluation of mathematical functions (e.g. Lisp, Scheme, Haskell, etc.)
- **Imperative**
  - Describes computation in terms of statements that change a program state (e.g. FORTRAN, BASIC, Pascal, C, etc.)
- **Logical (declarative)**
  - Expresses the logic of a computation without describing its control flow (e.g. Prolog)
- **Object oriented**
  - Uses "objects" – data structures consisting of data fields and methods together with their interactions – to design applications and computer programs (e.g. C++, Java, C#, Python, etc.)

Programming Language Paradigms

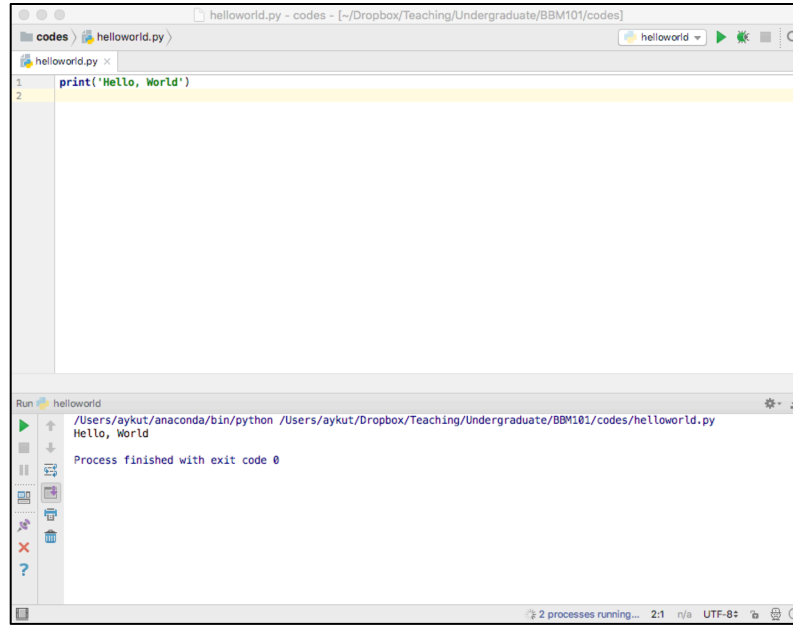
- **Functional**
  - Treats computation as the evaluation of mathematical functions (e.g. Lisp, Scheme, Haskell, etc.)
- **Imperative**
  - Describes computation in terms of statements that change a program state (e.g. FORTRAN, BASIC, Pascal, C, etc.)
- **Logical (declarative)**
  - Expresses the logic of a computation without describing its control flow (e.g. Prolog)
- **Object oriented**
  - Uses "objects" – data structures consisting of data fields and methods together with their interactions – to design applications and computer programs (e.g. C++, Java, C#, Python, etc.)

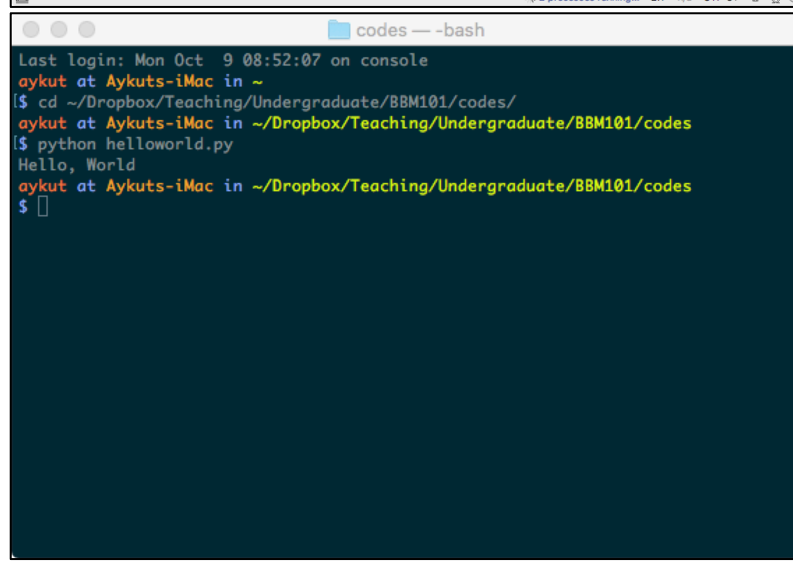
Lecture Overview

- Programming languages (PLs)
- Introduction to Python and Programming
- Control Flow

Programming in Python

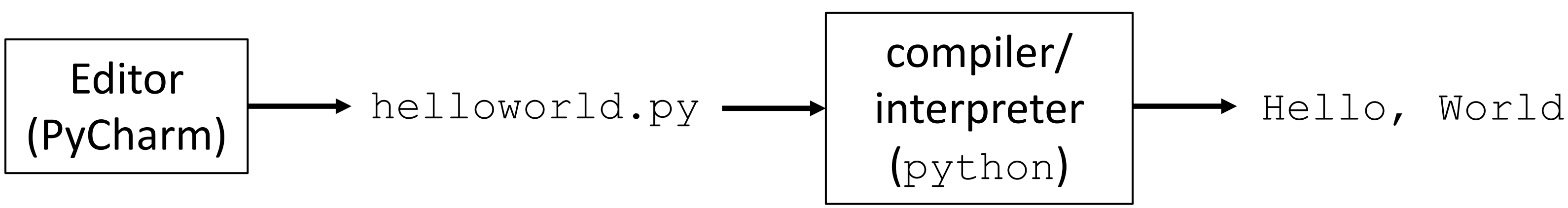
- Our programming environment
  - Python programming language
  - PyCharm, an integrated development environment (IDE)
  - Terminal





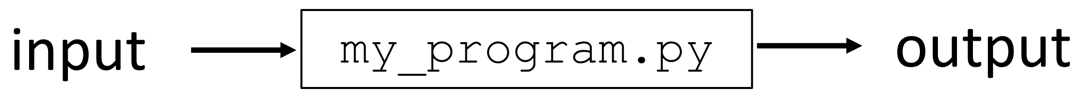
Programming in Python

- To program in Python
  - Compose a program by typing it into a file named, say, `helloworld.py`
  - Run (or execute) the program by typing `python helloworld.py` in the terminal window



Input and Output

- Bird's-eye view of a Python program



  - **Input types:** command-line arguments, standard input, file input
  - **Output types:** standard output, file output, graphical output, audio output

Input and Output

- Command-line arguments are the inputs we list after a program name when we run the program

```
$ python my_program.py arg_1 arg_2 ... arg_n
```

- The command-line arguments can be accessed within a program, such as `my_program.py` above, via the array (aka list) `sys.argv`[1] as `sys.argv[1], sys.argv[2], . . . , sys.argv[n]`

- The name of the program (`my_program.py`) is stored in `sys.argv[0]`

[1] The `sys` module provides access to variables and functions that interact with the Python interpreter

Input and Output

useargument.py

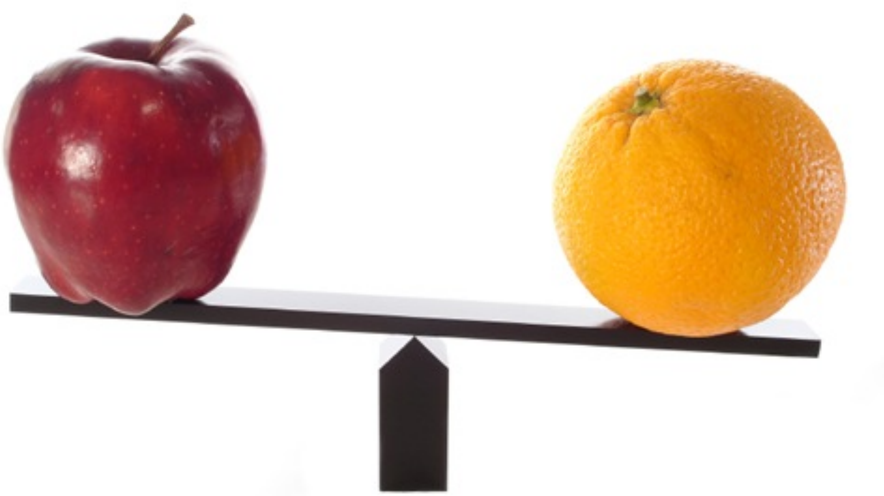
```
import sys

print('Hi, ', end='')
print(sys.argv[1], end='')
print('. How are you?')
```

```
$ python useargument.py Alice
Hi, Alice. How are you?
$ python useargument.py Bob
Hi, Bob. How are you?
$ python useargument.py Carol
Hi, Carol. How are you?
```

1. Python is like a calculator

2. A variable is a container

3. Different types cannot be compared



4. A program is a recipe

1. Python is Like a Calculator

You Type *Expressions*. Python Computes Their V*alues*.

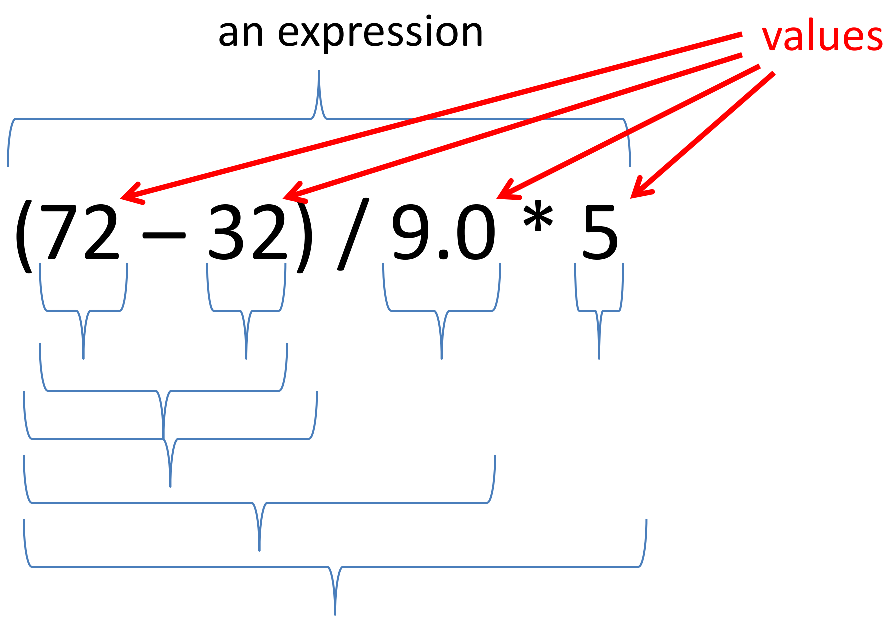
- 5
- 3+4
- 44/2
- 2**3
- 3*4+5*6
- (72 – 32) / 9 * 5

Python has a natural and well-defined set of precedence rules that fully specify the order in which the operators are applied in an expression

- For arithmetic operations, multiplication and division are performed before addition and subtraction
- When arithmetic operations have the same precedence, they are left associative, with the exception of the exponentiation operator **, which is right associative
- We can use parentheses to override precedence rules

An Expression is Evaluated From the Inside Out

- How many expressions are in this Python code?



(**72** – **32**) / **9.0** * **5**

(**40**) / **9.0** * **5**

40 / **9.0** * **5**

4.44 * **5**

22.2

Another Evaluation Example

(72 – 32) / (9.0 * 5)

(40) / (9.0 * 5)

40 / (9.0 * 5)

40 / (45.0)

40 / 45.0

.888

2. A Variable is a Container

A variable is a name associated with a data-type value

Variables Hold Values

- Recall variables from algebra:
  - Let x = 2 …
  - Let y = x …

- To assign a variable, use "*varname* = *expression*"

```
pi = 3.14
pi
var = 6*10**23
22 = x          # Error!
```

No output from an assignment statement

- Not all variable names are permitted!

  - Variable names must only be one word (as in no spaces)
  - Variable names must be made up of only letters, numbers, and underscore (_)
  - Variable names cannot begin with a number

Changing Existing Variables (“re-binding” or “re-assigning”)

```
x = 2
x
y = x
y
x = 5
x
y
```

- “**=**” in an assignment is not a promise of eternal equality
  - This is different than the mathematical meaning of “=”

- Evaluating an expression gives a new (copy of a) number, rather than changing an existing one

How an Assignment is Executed

1. Evaluate the right-hand side to a value
2. Store that value in the variable

```
x = 2
print(x)
y = x
print(y)
z = x + 1
print(z)
x = 5
print(x)
print(y)
print(z)
```

State of the computer:

x: 2 5
y: 2
z: 3

Printed output:

2
2
3
5
2
3

To visualize a program's execution:
http://pythontutor.com

More Expressions: Conditionals (value is `True` or `False`)

```
22 > 4      # condition, or conditional
22 < 4      # condition, or conditional
22 == 4     …
x = 100     # Assignment, not conditional!
22 = 4      # Error!
x >= 5
x >= 100
x >= 200
not True
not (x >= 200)
3<4 and 5<6
4<3 or 5<6
temp = 72
water_is_liquid = (temp > 32 and temp < 212)
```

Numeric operators: **+**, *****, ******

Boolean operators: **not**, **and**, **or**

Mixed operators: **<**, **>=**, **==**

More Expressions: strings

- A string represents **text**
  - `'Python'`
  - `myString = "BBM 101-Introduction to Programming"`
  - `""`

- Empty string is not the same as an unbound variable
  - `""` and ‘’ are the same

- We can specify tab, newline, backslash, and single quote characters using escape sequences `'\t'`, `'\n'`, `'\\'`, and `'\''`, respectively

Operations:

- **Length:**
  - `len(myString)`

- **Concatenation:**
  - `"Hacettepe" + " " + ' University'`

- **Containment/searching:**
  - `'a' in myString`
  - `"a" in myString`

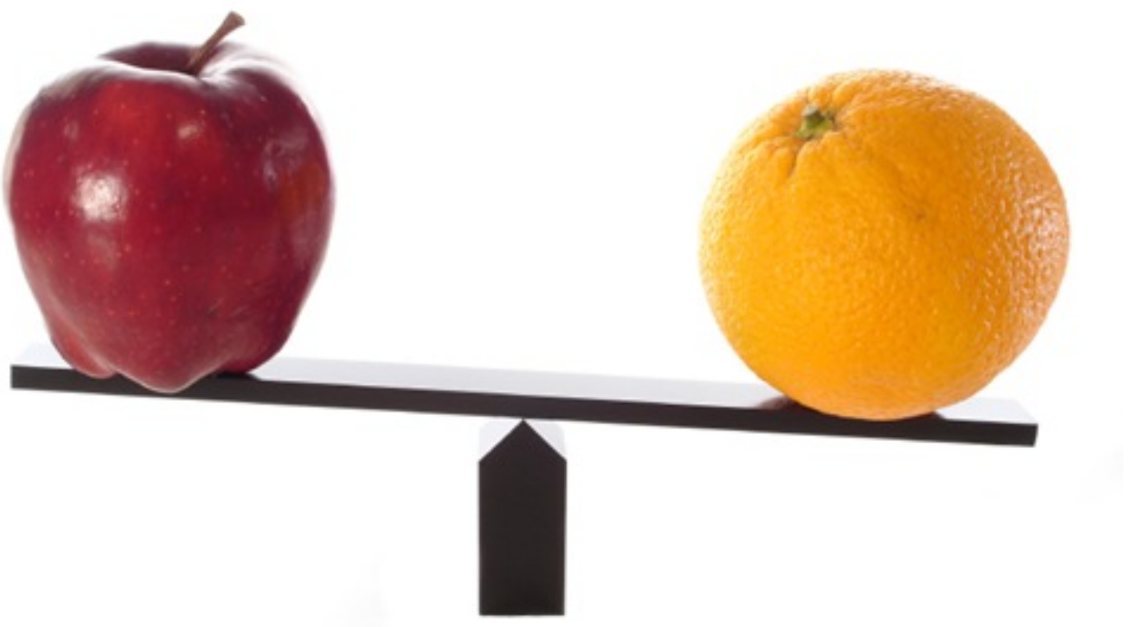
Strings

```
ruler1 = '1'
ruler2 = ruler1 + ' 2 ' + ruler1
ruler3 = ruler2 + ' 3 ' + ruler2
ruler4 = ruler3 + ' 4 ' + ruler3
print(ruler1)
print(ruler2)
print(ruler3)
print(ruler4)
```

```
1
1 2 1
1 2 1 3 1 2 1
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
```

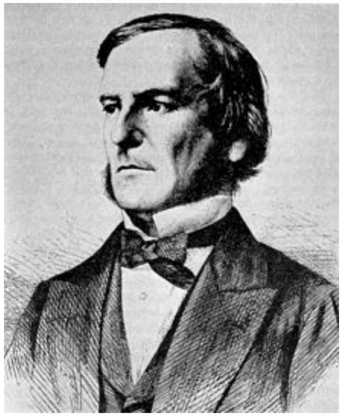
3. Different Types cannot be Compared

```
anInt = 2
aString = "Hacettepe"
anInt == aString                    # Error
```



Types of Values

- Integers (**`int`**): **`-22`**, **`0`**, **`44`**
  - Arithmetic is exact
  - Some funny representations: **`12345678901L`**
- Real numbers (**`float`**, for "floating point"): **`2.718`**, **`3.1415`**
  - Arithmetic is approximate, e.g., **`6.022*10**23`**
- Strings (**`str`**): **`"I love Python"`**, **`" "`**
- Truth values (**`bool`**, for "Boolean"): **`True`**, **`False`**



George Boole

Operations Behave differently on Different Types

```
3.0 + 4.0
3 + 4
3 + 4.0
"3" + "4"          # Concatenation
3 + "4"            # Error
3 + True           # Error
```

Moral: Python only *sometimes* tells you when you do something that does not make sense.

Operations on Different Types

	Python 3.5	Python 2.x
`15.0 / 4.0`	3.75	3.75
`15 / 4`	3.75	3
`15.0 / 4`	3.75	3.75
`15 / 4.0`	3.75	3.75

`15.0 // 4.0`	3.0
`15 // 4`	3
`15.0 // 4`	3.0
`15 // 4.0`	3.0

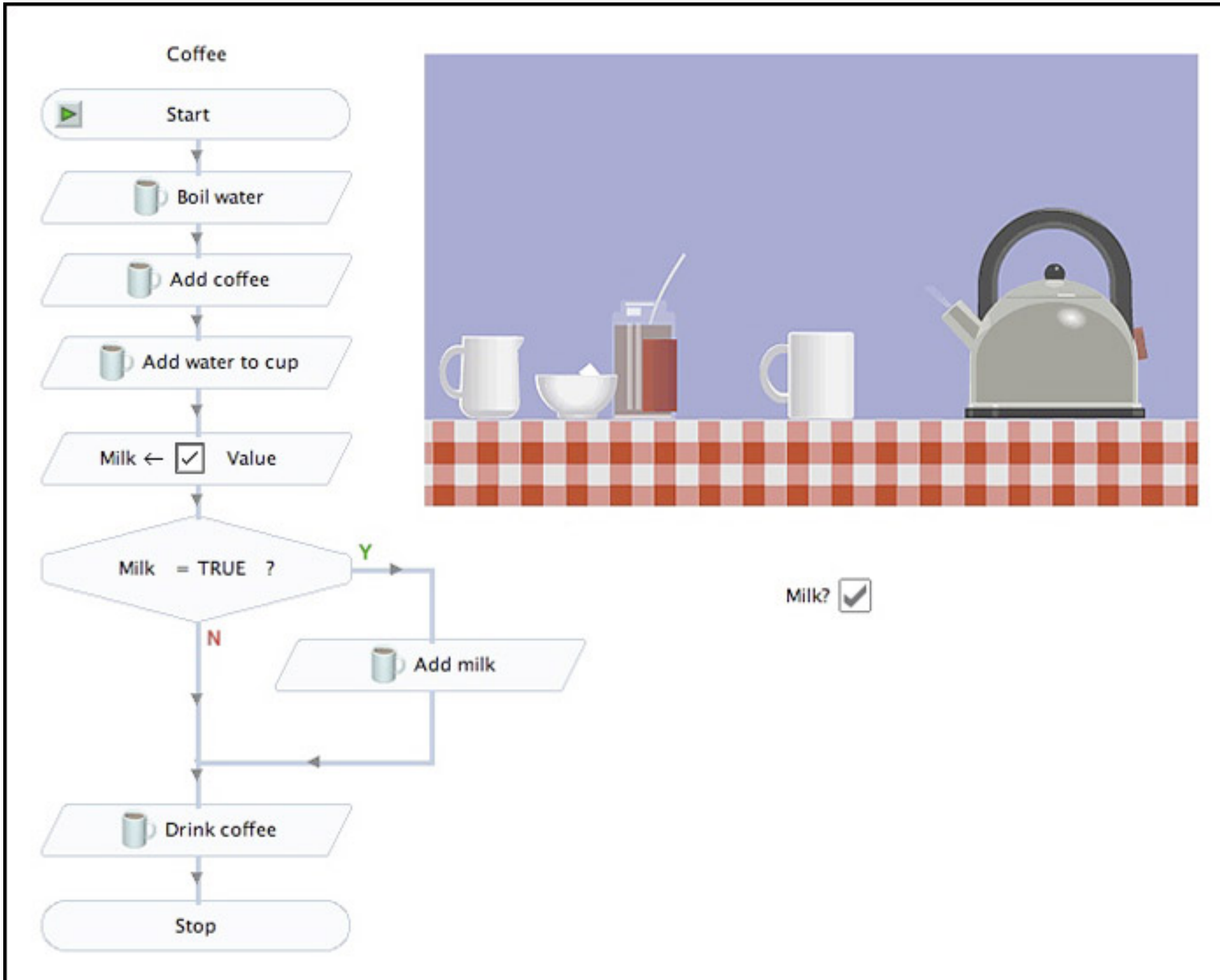
Before Python version 3.5, operand used to determine the type of division.

/ : Division
//: Integer Division

Type Conversion

`float(15)`	15.0
`int(15.0)`	15
`int(15.5)`	15
`int("15")`	15
`str(15.5)`	15.5
`float(15) / 4`	3.75

A Program is a Recipe



Design the Algorithm Before Coding

- We should think (design the algorithm) before coding

- Algorithmic thinking is the logic. Also, called problem solving

- Coding is the syntax

- Make this a habit

- Some students do not follow this practice and they get challenged in all their courses and careers!

What is a Program?

- A program is a sequence of instructions
- The computer executes one after the other, as if they had been typed to the interpreter
- Saving your work as a program is better than re-typing from scratch

```
x = 1
y = 2
x + y
print(x + y)
print("The sum of", x, "and", y, "is", x+y)
```

The `print()` Statement

- The **`print`** statement always prints one line
  - The next print statement prints below that one
- Write 0 or more expressions after **`print`**, separated by commas
  - In the output, the values are separated by spaces
- Examples:

```
x = 1
y = 2
print(3.1415)
print(2.718, 1.618)
print()
print(20 + 2, 7 * 3, 4 * 5)
print("The sum of", x, end="")
print(" and", y, "is", x+y)
```

```
3.1415
2.718 1.618

22 21 20
The sum of 1 and 2 is 3
```

To avoid newline

Exercise: Convert Temperatures

- Make a temperature conversion chart as the following
- Fahrenheit to Centigrade, for Fahrenheit values of: -40, 0, 32, 68, 98.6, 212
- C = (F - 32) × 5/9
- Output:

```
Fahrenheit Centigrade
-40 -40.0
0 -17.7778
32 0.0
68 20.0
98.6 37.0
212 100.0
```

- You have created a Python program!
- (It doesn't have to be this tedious, and it won't be.)

Expressions, Statements, and Programs

- An expression evaluates to a value

```
3 + 4
pi * r**2
```

- A statement causes an effect

```
pi = 3.14159
print(pi)
```

- Expressions appear within other expressions and within statements

```
(fahr – 32) * (5.0 / 9)
print(pi * r**2)
```

- A statement may *not* appear within an expression

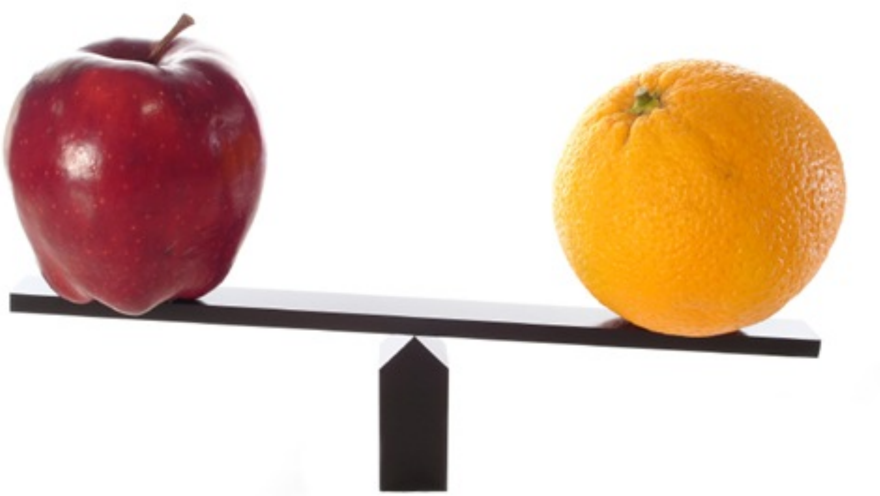
```
3 + print(pi)     # Error!
```

- A program is made up of statements
  - A program should do something or communicate information

1. Python is like a calculator

2. A variable is a container

3. Different types cannot be compared



4. A program is a recipe


CORNBREAD

Colvin Run Mill Corn Bread

1 cup cornmeal
1 cup flour
½ teaspoon salt
4 teaspoons baking powder
3 tablespoons sugar
1 egg
1 cup milk
¼ cup shortening (soft) or vegetable oil

Mix together the dry ingredients. Beat together the egg, milk and shortening/oil. Add the liquids to the dry ingredients. Mix quickly by hand. Pour into greased 8x8 or 9x9 baking pan. Bake at 425 degrees for 20-25 minutes.

Programming Languages

- A programming language is a "language" to write programs in, such as Python, C, C++, Java
- The concept of programming languages are quite similar
- Python:

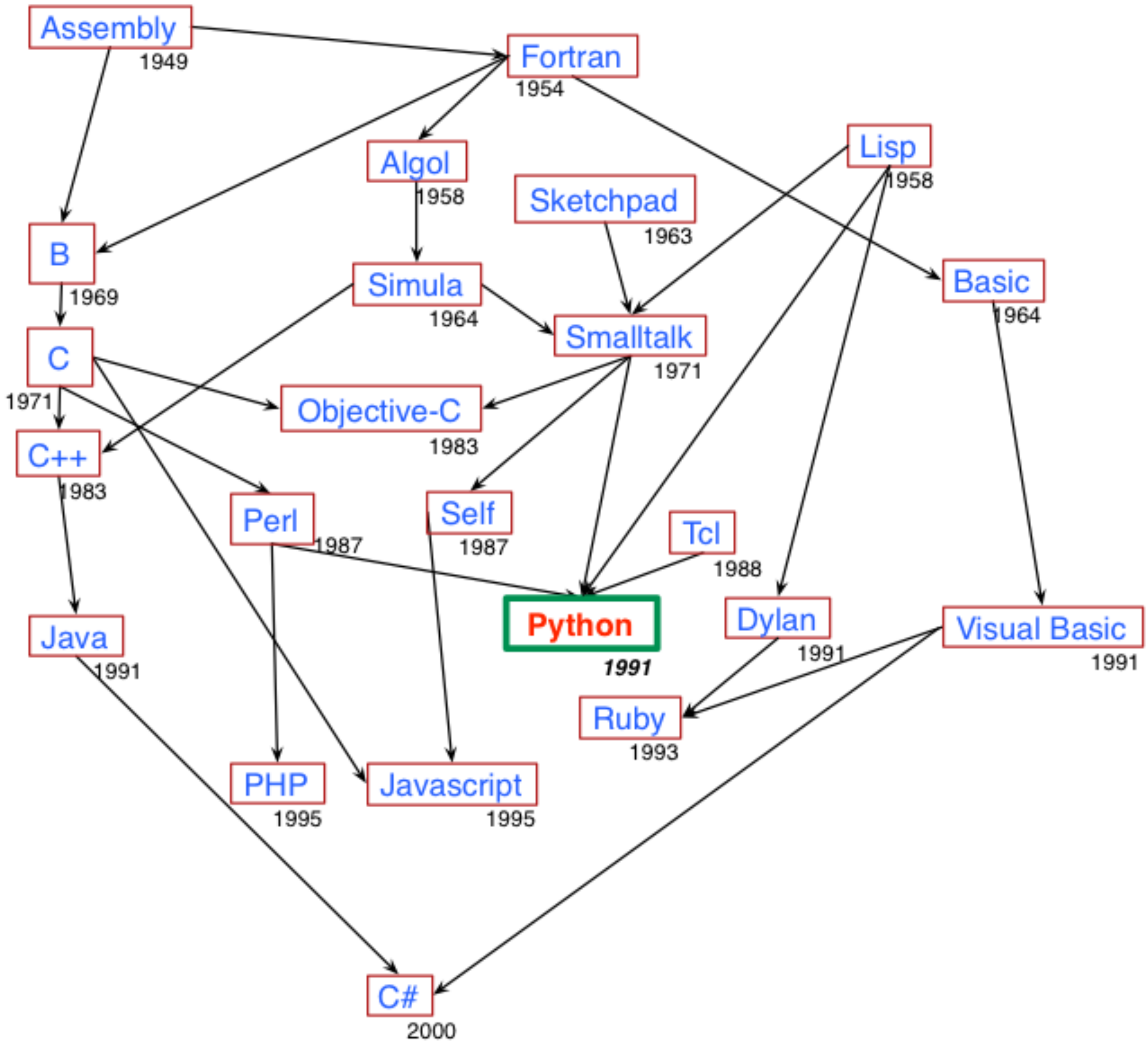
```python
print("Hello, World!")
```

- Java:

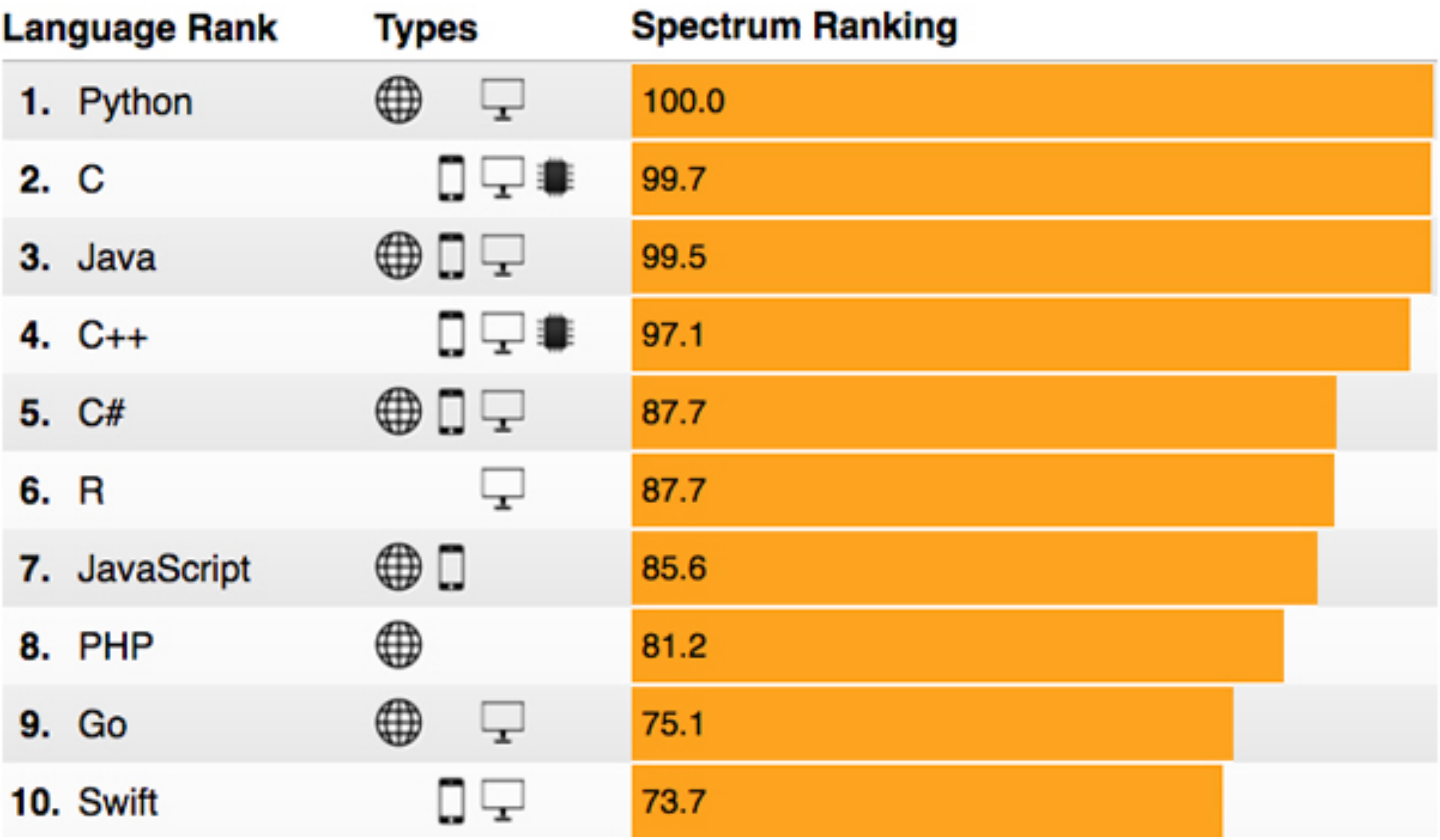
```java
public static void main(String[] args) {
    System.out.println("Hello, World!");
}
```

- Python is simpler! That's why we are learning it first ☺

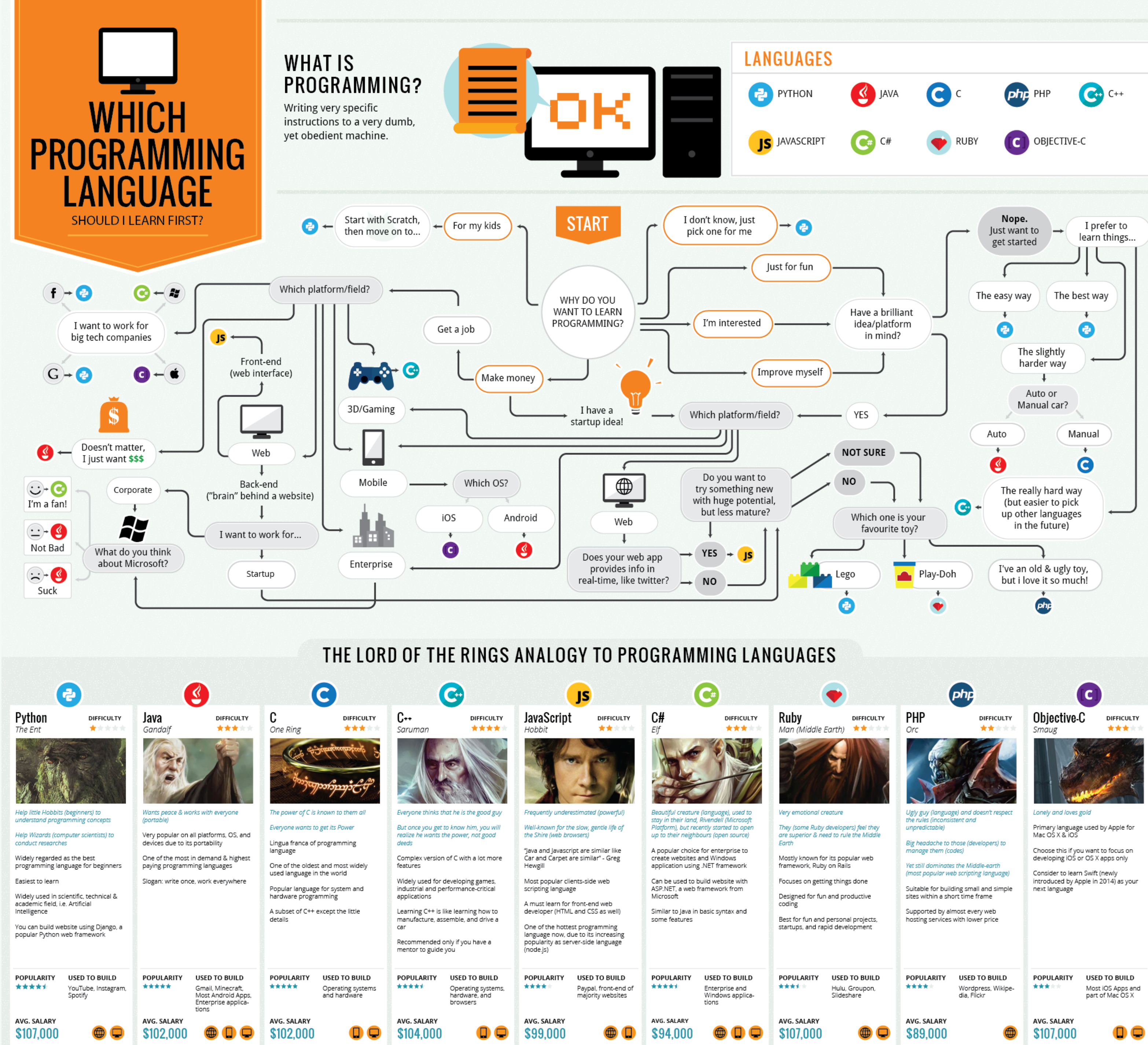
Evolution of Programming Languages



The 2017 Top Programming Languages



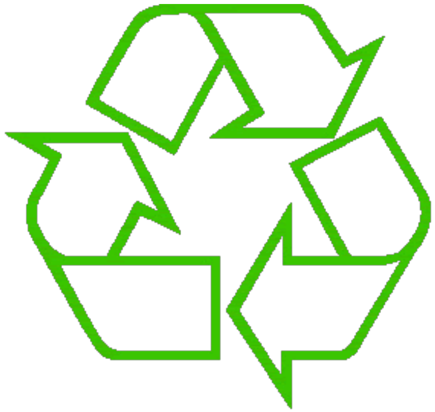
- https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages



http://carlcheo.com/startcoding

Lecture Overview

- Programming languages (PLs)
- Introduction to Python and Programming
- Control Flow



Repeating yourself

Making decisions

Temperature Conversion Chart

Recall the exercise from the previous lecture

```
fahr = 30
cent = (fahr -32)/9.0*5
print(fahr, cent)
fahr = 40
cent = (fahr -32)/9.0*5
print(fahr, cent)
fahr = 50
cent = (fahr -32)/9.0*5
print(fahr, cent)
fahr = 60
cent = (fahr -32)/9.0*5
print(fahr, cent)
fahr = 70
cent = (fahr -32)/9.0*5
print(fahr, cent)
Print("All done")
```

Output:

```
30 -1.11
40 4.44
50 10.0
60 15.55
70 21.11
All done
```

Temperature Conversion Chart

A better way to repeat yourself:

for loop

loop variable or iteration variable

A list

Colon is required

Loop *body* is indented

```
for f in [30,40,50,60,70]:
    print(f, (f-32)/9.0*5)
print("All done")
```

Execute the body 5 times:

- once with f = 30
- once with f = 40
- once with f = 50
- once with f = 60
- once with f = 70

Indentation is significant

Output:

```
30 -1.11
40 4.44
50 10.0
60 15.55
70 21.11
All done
```

How a Loop is Executed: Transformation Approach

Idea: convert a **`for`** loop into something we know how to execute

1. Evaluate the sequence expression
2. Write an assignment to the loop variable, for each sequence element
3. Write a copy of the loop after each assignment
4. Execute the resulting statements

```
for i in [1,4,9]:
  print(i)
```

→

```
i = 1
print(i)
i = 4
print(i)
i = 9
print(i)
```

State of the computer:

i: 9

Printed output:

1
4
9

How a Loop is Executed: Direct Approach

1. Evaluate the sequence expression
2. While there are sequence elements left:
   a) Assign the loop variable to the next remaining sequence element
   b) Execute the loop body

Current location in list

```
for i in [1,4,9]:
  print(i)
```

State of the computer:

Printed output:

1
4
9

The Body can be Multiple Statements

Execute whole body, then execute whole body again, etc.

```
for i in [3,4,5]:
    print("Start body")
    print(i)
    print(i*i)
```

loop body:
3 statements

Output:	NOT:
Start body	~~Start body~~
3	~~Start body~~
9	~~Start body~~
Start body	~~3~~
4	~~4~~
16	~~5~~
Start body	~~9~~
5	~~16~~
25	~~25~~

Convention: often use *i* or *j* as loop variable if values are integers

This is an exception to the rule that variable names should be descriptive

Indentation in Loop is Significant

- Every statement in the body must have exactly the same indentation
- That's how Python knows where the body ends

```
for i in [3,4,5]:
    print("Start body")
     print(i)
    print(i*i)
```

Error!

- Compare the results of these loops:

```
for f in [30,40,50,60,70]:
    print(f, (f-32)/9.0*5)
print("All done")

for f in [30,40,50,60,70]:
    print(f, (f-32)/9.0*5)
    print("All done")
```

The Body can be Multiple Statements

How many statements does this loop contain?

```
for i in [0,1]:
    print("Outer", i)
    for j in [2,3]:
        print(" Inner", j)
        print("  Sum", i+j)
    print("Outer", i)
```

"nested" loop body: 2 statements

loop body: 3 statements

What is the output?

Output:
Outer 0
  Inner 2
    Sum 2
  Inner 3
    Sum 3
Outer 0
Outer 1
  Inner 2
    Sum 3
  Inner 3
    Sum 4
Outer 1

Understand Loops Through the Transformation Approach

Key idea:

1. Assign each sequence element to the loop variable
2. Duplicate the body

```
for i in [0,1]:
  print("Outer", i)
  for j in [2,3]:
    print(" Inner", j)
```

```
i = 0
print("Outer", i)
for j in [2,3]:
  print(" Inner", j)
i = 1
print("Outer", i)
for j in [2,3]:
  print(" Inner", j)
```

```
i = 0
print("Outer", i)
j = 2
print(" Inner", j)
j = 3
print(" Inner", j)
i = 1
print("Outer", i)
for j in [2,3]:
  print(" Inner", j)
```

Fix This Loop

```
# Goal:  print 1, 2, 3, …, 48, 49, 50
for tens_digit in [0, 1, 2, 3, 4]:
  for ones_digit in [1, 2, 3, 4, 5, 6, 7, 8, 9]:
    print(tens_digit * 10 + ones_digit)
```

What does it actually print?

How can we change it to correct its output?

Moral: Watch out for *edge conditions* (beginning or end of loop)

Some Fixes

```
# Goal:  print 1, 2, 3, …, 48, 49, 50

for tens_digit in [0, 1, 2, 3, 4]:
  for ones_digit in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
    print(tens_digit * 10 + ones_digit + 1)

for tens_digit in [0, 1, 2, 3, 4]:
  for ones_digit in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    print(tens_digit * 10 + ones_digit)

for tens_digit in [1, 2, 3, 4]:
  for ones_digit in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
    print(tens_digit * 10 + ones_digit)
print 50
```

- Analyze each of the above

Test Your Understanding of Loops

Output:

Puzzle 1:

```
for i in [0,1]:
  print(i)
print(i)
```

Output:

```
0
1
1
```

Puzzle 2:

```
i = 5
for i in []:
  print(i)
```

Output: (no output)

Puzzle 3:

```
for i in [0,1]:
  print("Outer", i)
  for i in [2,3]:
    print(" Inner", i)
  print("Outer", i)
```

Reusing loop variable (don't do this!)

inner loop body

outer loop body

Output:

```
Outer 0
  Inner 2
  Inner 3
Outer 3
Outer 1
  Inner 2
  Inner 3
Outer 3
```

The Range Function

As an implicit list:

```
for i in range(5):
    … body …
```

The list [0,1,2,3,4]

Upper limit (*exclusive*)

range(5) = [0,1,2,3,4]

Lower limit (*inclusive*)

range(1,5) = [1,2,3,4]

step (distance between elements)

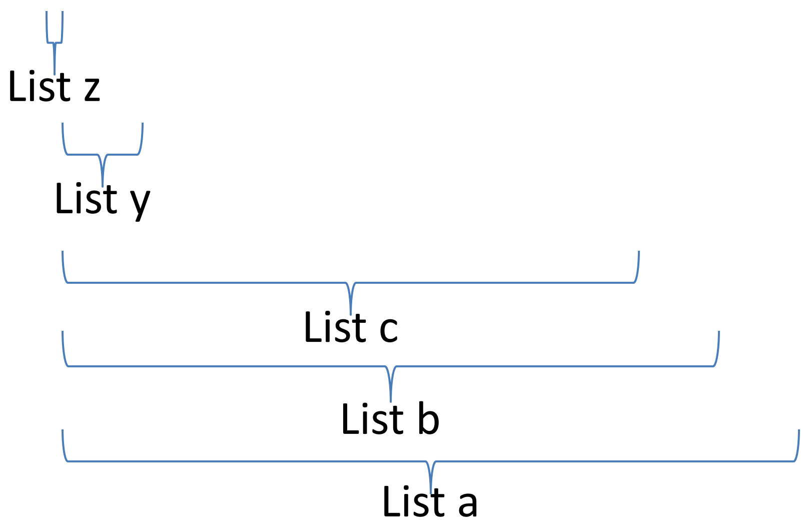
range(1,10,2) = [1,3,5,7,9]

Decomposing a List Computation

- To compute a value for a list:
  - Compute a partial result for all but the last element
  - Combine the partial result with the last element

Example: sum of a list:

[3, 1, 4, 1, 5, 9, 2, 6, 5]



sum(List a) = sum(List b) + 5
sum(List b) = sum(List c) + 6
…
sum(List y) = sum(List z) + 3
sum(empty list) = 0

How to Process a List: One Element at a Time

- A common pattern when processing a list:

```
result = initial_value
for element in list:
  result = updated result
use result
```

```
# Sum of a list
result = 0
for element in mylist:
  result = result + element
print result
```

- `initial_value` is a correct result for an empty list
- As each element is processed, `result` is a correct result for a prefix of the list
- When all elements have been processed, `result` is a correct result for the whole list

Some Loops

```
# Sum of a list of values, what values?
result = 0
for element in range(5): # [0,1,2,3,4]
  result = result + element
print("The sum is: " + str(result))
```

The sum is: 10

```
# Sum of a list of values, what values?
result = 0
for element in range(5,1,-1):
  result = result + element
print("The sum is:", result)
```

5, 4, 3, 2
The sum is: 14

```
# Sum of a list of values, what values?
result = 0
for element in range(0,8,2):
  result = result + element
print("The sum is:", result)
```

0, 2, 4, 6
The sum is: 12

```
# Sum of a list of values, what values?
result = 0
size = 5
for element in range(size):
  result = result + element
print("When size = " + str(size) + ", the result is " + str(result))
```

0, 1, 2, 3, 4
When size = 5, the result is 10

`divisorpattern.py`: Accept integer command-line argument *n*. Write to standard output an *n*-by-*n* table with an asterisk in row i and column j if either i divides j or j divides i.

```python
import sys

n = int(sys.argv[1])
for i in range(1, n + 1):
    for j in range(1, n + 1):
        if (i % j == 0) or (j % i == 0):
            print('* ', end='')
        else:
            print('  ', end='')
    print(i)
```

```
$ python divisorpattern.py 3
* * * 1
* *   2
*   * 3
```

```
$ python divisorpattern.py 10
* * * * * * * * * * 1
* *   *   *   *   * 2
*   *     *     *   3
* *   *       *     4
*       *         * 5
* * *     *         6
*           *       7
* *   *       *     8
*   *           *   9
* *     *         * 10
```

Variable trace (`n = 3`)

```
i    j    output
----------------
1    1    '* '
1    2    '* '
1    3    '* 1\n'
2    1    '* '
2    2    '* '
2    3    '  2\n'
3    1    '* '
3    2    '  '
3    3    '* 3\n'
```

Examples of List Processing

```
result = initial_value
for element in list:
  result = updated result
```

- Product of a list:

```
result = 1
for element in mylist:
  result = result * element
```

- Maximum of a list:

```
result = mylist[0]
for element in mylist:
  result = max(result, element)
```

The first element of the list (counting from zero)

- Approximate the value 3 by 1 + 2/3 + 4/9 + 8/27 + 16/81 + … = $(2/3)^0 + (2/3)^1 + (2/3)^2 + (2/3)^3 + \ldots + (2/3)^{10}$

```
result = 0
for element in range(11):
  result = result + (2.0/3.0)**element
```

Exercise with Loops

- Write a simple program to add values between two given inputs a, b
- e.g., if a=5, b=9, it returns sum of (5+6+7+8+9)
- Hint: we did some 'algorithmic thinking' and 'problem solving' here!

```
a=5
b=9
total = 0
for x in range(a, b+1):
    total += x
print(total)
```

Another Type of Loops

- The **`while`** loop is used for repeated execution as long as an expression is true

```
n = 100
s = 0
counter = 1
while counter <= n:
    s = s + counter
    counter += 1

print("Sum of 1 until %d: %d" % (n,s))
```

```
Sum of 1 until 100: 5050
```

Making Decisions

- How do we compute absolute value?

```
abs(5) = 5
abs(0) = 0
abs(-22) = 22
```

Absolute Value Solution

<u>**If**</u> *the value is negative*, negate it.

<u>**Otherwise**</u>, use the original value.

```
val = -10

# calculate absolute value of val
if val < 0:
    result = - val
else:
    result = val

print(result)
```

Another approach
that does the same thing
without using **`result`**:

```
val = -10

if val < 0:
    print(- val)
else:
    print(val)
```

In this example, **`result`** will always be assigned a value.

Absolute Value Solution

As with loops, a <u>sequence of statements</u> could be used in place of a single statement inside an if statement:

```
val = -10

# calculate absolute value of val
if val < 0:
    result = - val
    print("val is negative!")
    print("I had to do extra work!")
else:
    result = val
    print("val is positive")
print(result)
```

Absolute Value Solution

What happens here?

```
val = 5

# calculate absolute value of val
if val < 0:
    result = - val
    print("val is negative!")
else:
    for i in range(val):
        print("val is positive!")
    result = val
print(result)
```

Another if

It is **<u>not required that anything happens</u>**…

```
val = -10

if val < 0:
     print("negative value!")
```

What happens when val = 5?

The if Body can be Any Statements

then clause / else clause (t / f)

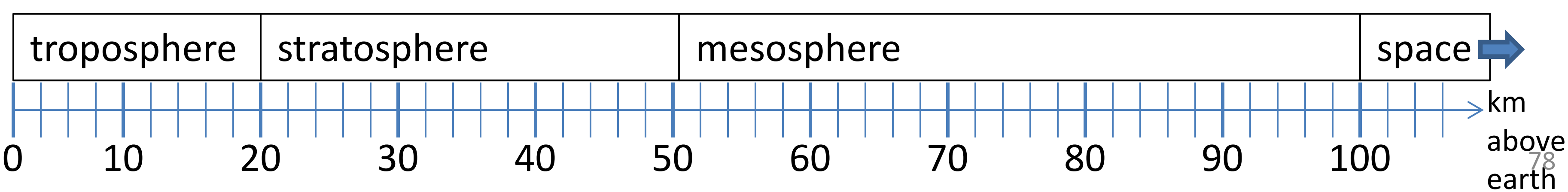
```
# height is in km
if height > 100:
  print("space")
else:
  if height > 50:
    print("mesosphere")
  else:
    if height > 20:
      print("stratosphere")
    else:
      print("troposphere")
```

Execution gets here only if "height > 100" is false

Execution gets here only if "height > 100" is false AND "height > 50" is true

Written differently, but more efficient!

```
# height is in km
if height > 100:
  print("space")
elif height > 50:
  print("mesosphere")
elif height > 20:
  print("stratosphere")
else:
  print("troposphere")
```



Version 1

```
# height is in km
if height > 100:
    print("space")
else:
    if height > 50:
        print("mesosphere")
    else:
        if height > 20:
            print("stratosphere")
        else:
            print("troposphere")
```

then clause

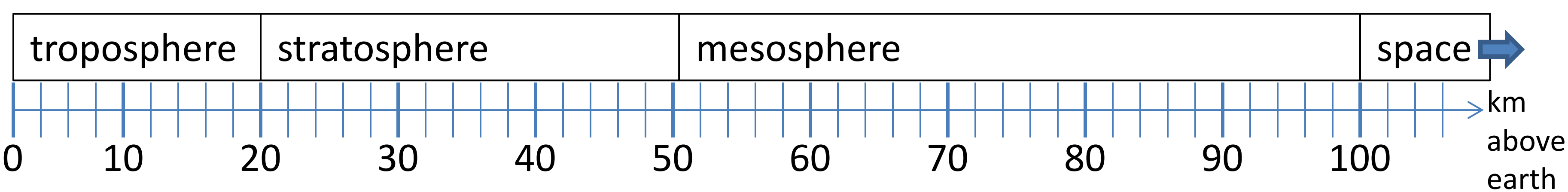
else clause

t

e

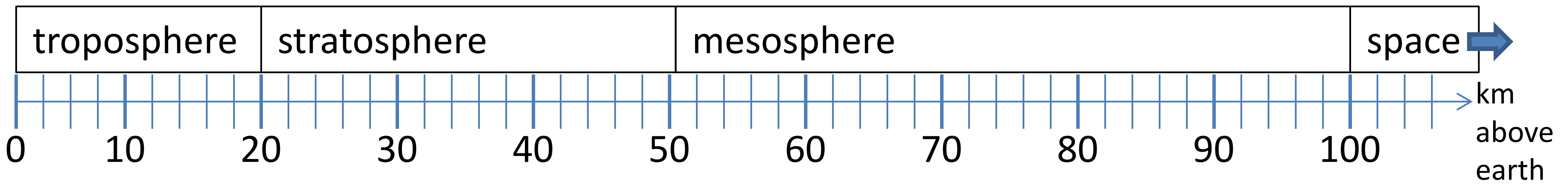
Execution gets here only if “height <= 100” is true

Execution gets here only if “height <= 100” is true AND “height > 50” is true



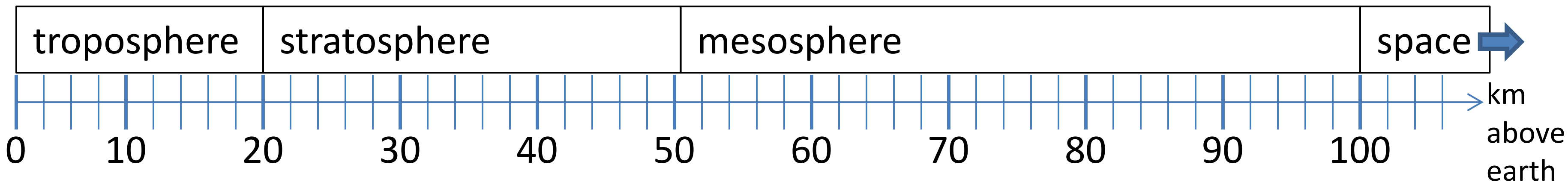
Version 1

```
# height is in km
if height > 100:
  print("space")
else:
  if height > 50:
    print("mesosphere")
  else:
    if height > 20:
      print("stratosphere")
    else:
      print("troposphere")
```



Version 2

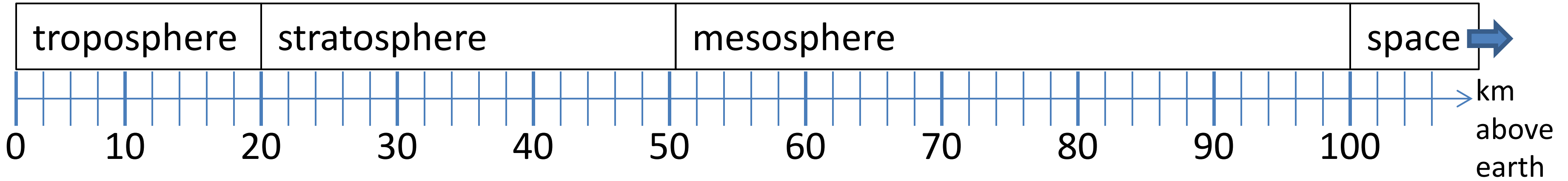
```
if height > 50:
  if height > 100:
    print("space")
  else:
    print("mesosphere")
else:
  if height > 20:
    print("stratosphere")
  else:
    print("troposphere")
```



Version 3

```
if height > 100:
  print("space")
elif height > 50:
  print("mesosphere")
elif height > 20:
  print("stratosphere")
else:
  print("troposphere")
```

ONE of the print statements is guaranteed to execute: whichever condition it encounters **first** that is true

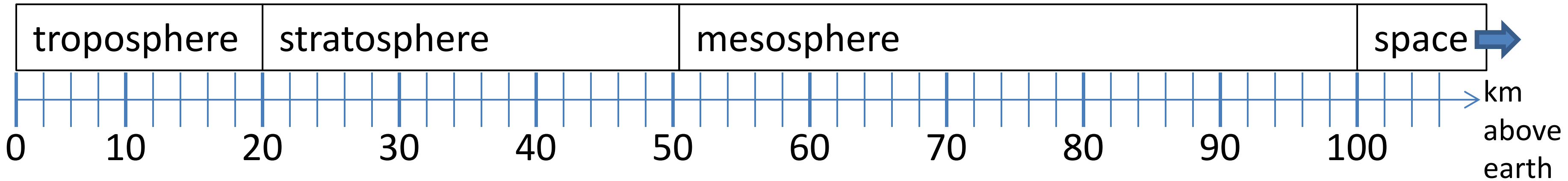


Order Matters

```
# version 3
if height > 100:
  print("space")
elif height > 50:
  print("mesosphere")
elif height > 20:
  print("stratosphere")
else:
  print("troposphere")
```

```
# broken version 3
if height > 20:
  print("stratosphere")
elif height > 50:
  print("mesosphere")
elif height > 100:
  print("space")
else:
  print("troposphere")
```

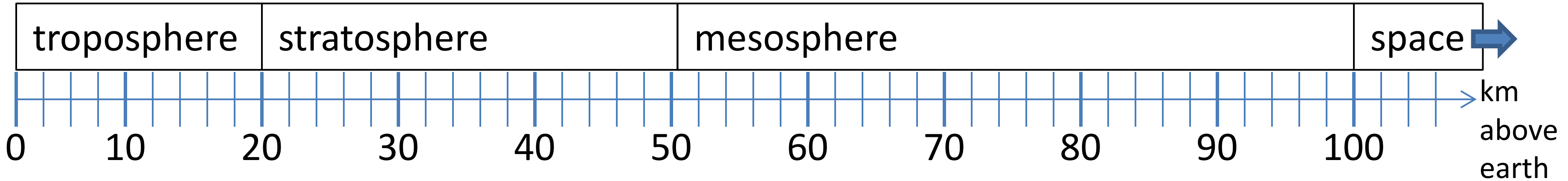
Try height = 72 on both versions, what happens?



Version 3

```
# incomplete version 3
if height > 100:
  print("space")
elif height > 50:
  print("mesosphere")
elif height > 20:
  print("stratosphere")
```

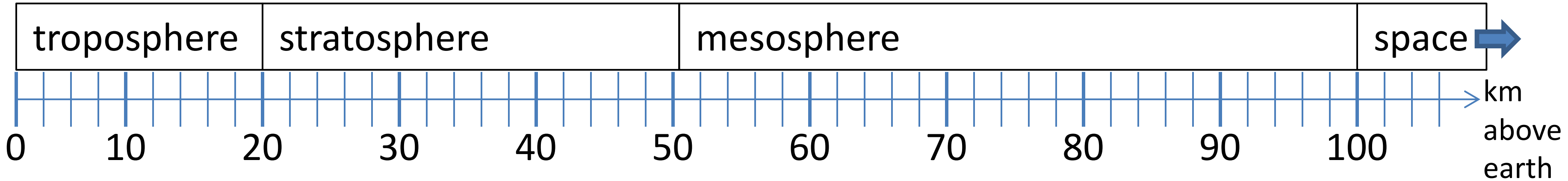
In this case it is possible that nothing is printed at all, when?



What Happens Here?

```
# height is in km
if height > 100:
  print("space")
if height > 50:
  print("mesosphere")
if height > 20:
  print("stratosphere")
else:
  print("troposphere")
```

Try height = 72



The then Clause *or* the else Clause is Executed


```
speed = 65
limit = 70
if speed <= limit:
    print("Good job, safe driver!")
else:
    print("You owe $", speed/fine)
```

What if we change speed to 50?

The `break` Statement

- The **`break`** statement terminates the current loop and resumes execution at the next statement

```
for letter in 'hollywood':
   if letter == 'l':
      break
   print ('Current Letter :', letter)
```

```
Current Letter : h
Current Letter : o
```

The `continue` Statement

- The **`continue`** statement in Python returns the control to the beginning of the while loop.

```
for letter in 'hollywood':
    if letter == 'l':
        continue
    print ('Current Letter :', letter)
```

```
Current Letter : h
Current Letter : o
Current Letter : y
Current Letter : w
Current Letter : o
Current Letter : o
Current Letter : d
```