WELCOME!

(download slides and .py files from Stellar to follow along)

6.0001 LECTURE 1

John Guttag

TODAY

- Course info
- What is computation
- Python basics
  - mathematical operations
  - python variables and types
- Flow of control
- NOTE: **slides and code files up before each lecture**
  - highly encourage you to download them before lecture
  - take notes and run code files when I do
  - bring computers to answer **in-class practice exercises!**

COURSE INFO

- Stellar course site
  - https://stellar.mit.edu/S/course/6/fa17/6.0001
  - https://stellar.mit.edu/S/course/6/fa17/6.00
  - links to Piazza, MITx, Calendar, Grades, details on course policies
- Email staff asap if have problems with schedule
- Course uses **Python 3** (do not use Python 2)
- Prerequisites
  - High school math
  - MIT-caliber brain
  - Little or no programming experience

COURSE POLICIES

- Collaboration
  - Okay
    - Helping others debug
    - Discussing general attack on problem
  - Not okay
    - Copying code (from others in class or previous years)
    - Side-by-side coding
  - Provide names of all "collaborators" on submission
  - We will be running a code similarity program on all psets
- Extensions
  - **No extensions**
  - **Late days**, see course website for details
  - **Drop and roll** (next slide)

Grading, Problem Sets and Finger Exercises

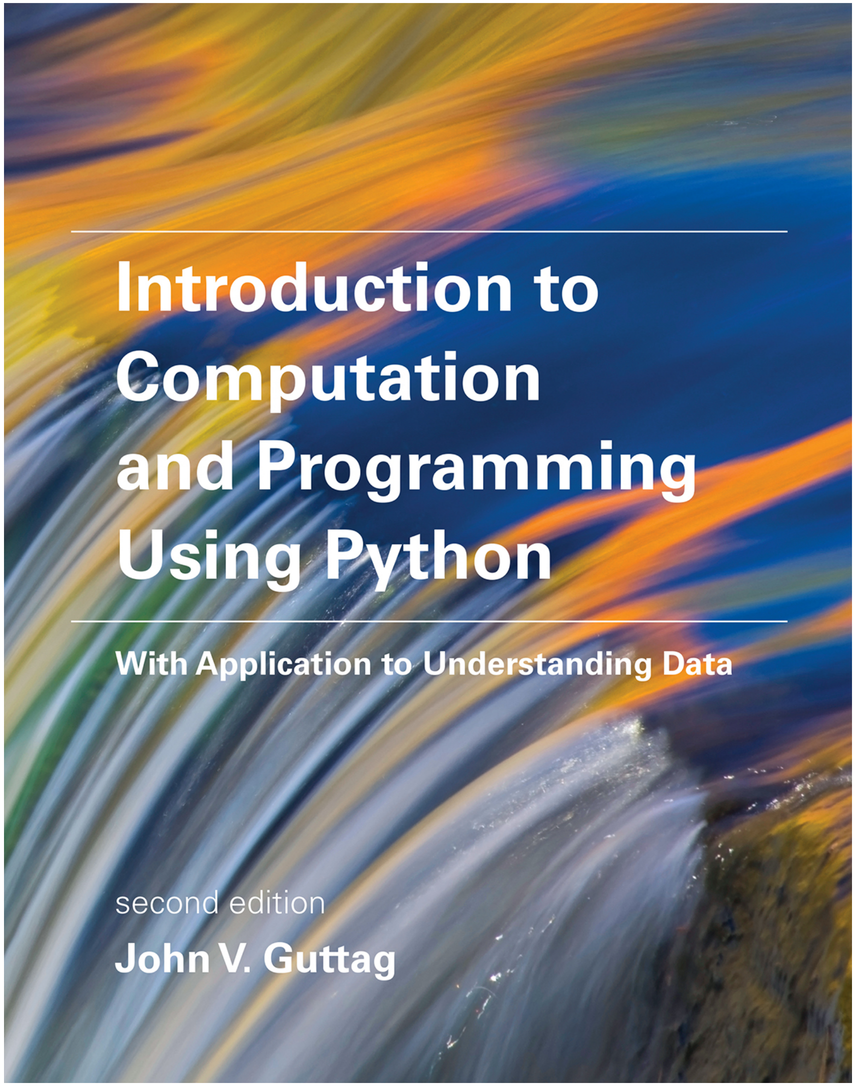
- Problem sets
  - 30% of final grade
  - 6.0001
    - 5 problem sets
    - If it would benefit you we will drop up to two PS grades and roll the points into the final
  - 6.00
    - 10 problem sets
    - If it would benefit you we will drop up to two PS grades *from each half* of the subject and roll the points into the final
- Finger exercises
  - 10% of final grade for mandatory finger exercises

Grading, Exams and Quizzes

- Final exam 40% (assuming no roll over)
- 6.00: Midterm for 20% of grade
- 6.0001: 3 micro quizzes for 20% of grade
  - Count only best 2 of 3
  - 15-20 minutes long and given at end of lecture
    - Must have computer with wireless connection
    - No conflict micro quizzes given
    - If it would benefit you we will drop one or both of the micro quizzes and roll the points into the final
- Exams will cover material from lectures, problem sets, and assigned readings

Assigned Reading

- Chapter 1
- Sections 2.1 – 2.3



https://mitpress.mit.edu/sites/default/files/Guttag_errata_revised_083117.pdf

Review Sessions

- Most Fridays
- Not mandatory

PROBLEM SETS

- Up on Stellar weekly, hand in online
- Score based on 2 components
  - how many **test cases you pass** (calculated automatically)
  - **checkoff for code style and explanation of code**
- Checkoffs starting with pset 1
  - Monday-Wednesday during office hours for the 10 days following the initial due date

Fast-paced Subject

- Position yourself to succeed!
  - **Read psets when they come out**
  - Save late days for emergency situations
  - Don't rely on rolling things over
- Learning to program
  - Can't passively absorb programming as a skill
  - Download code before lecture and follow along
  - Do MITx finger exercises
  - **Get help early**
    - Piazza, office hours, HKN tutoring: https://hkn.scripts.mit.edu/tutoring/
- Have fun

TOPICS

- 6.0001
  - Solving problems using **computation**
  - **Python** programming language
  - **Organizing modular programs**
  - Some simple but important **algorithms**
  - Algorithmic **complexity**
- 6.0002
  - Using computation to **model** the world
  - **Simulation** models
  - Understanding **data**

TYPES OF KNOWLEDGE

- **Declarative knowledge** is **statements of fact**
  - Someone will eat a candy during class
- **Imperative knowledge** is a **recipe** or “how-to”
  - (1) Walk to front of class
  - (2) Pick up candy
  - (3) Walk back to seat
  - (4) Unwrap candy
  - (5) Place candy in mouth
  - etc.
- Programming is about writing recipes to generate facts

A NUMERICAL EXAMPLE

- Square root of a number `x` is `y` such that `y*y = x`
- Start with a **guess**, `g`
  1) If `g*g` is **close enough** to `x`, stop and say `g` is the answer
  2) Otherwise make a **new guess** by averaging `g` and `x/g`
  3) Using the new guess, **repeat** process until close enough
- Let's try it for x = 16 and an initial guess of 3

g	g*g	x/g	(g+x/g)/2
3	9	16/3	4.17

A NUMERICAL EXAMPLE

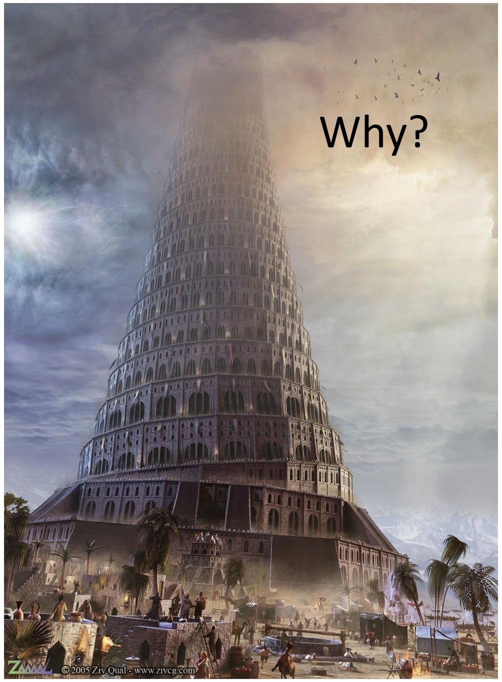
- Square root of a number `x` is `y` such that `y*y = x`
- Start with a **guess**, `g`
  1) If `g*g` is **close enough** to `x`, stop and say `g` is the answer
  2) Otherwise make a **new guess** by averaging `g` and `x/g`
  3) Using the new guess, **repeat** process until close enough
- Let’s try it for x = 16 and an initial guess of 3

g	g*g	x/g	(g+x/g)/2
3	9	16/3	4.17
4.17	17.36	3.837	4.0035

A NUMERICAL EXAMPLE

- Square root of a number `x` is `y` such that `y*y = x`
- Start with a **guess**, `g`
  1) If `g*g` is **close enough** to `x`, stop and say `g` is the answer
  2) Otherwise make a **new guess** by averaging `g` and `x/g`
  3) Using the new guess, **repeat** process until close enough
- Let's try it for x = 16 and an initial guess of 3

g	g*g	x/g	(g+x/g)/2
3	9	16/3	4.17
4.17	17.36	3.837	4.0035
4.0035	16.0277	3.997	4.000002

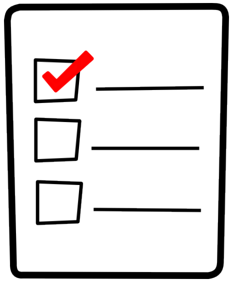


What We Have Here is an Algorithm

1) Sequence of simple **steps**

2) **Flow of control** process that specifies when each step is executed

3) A means of determining **when to stop**

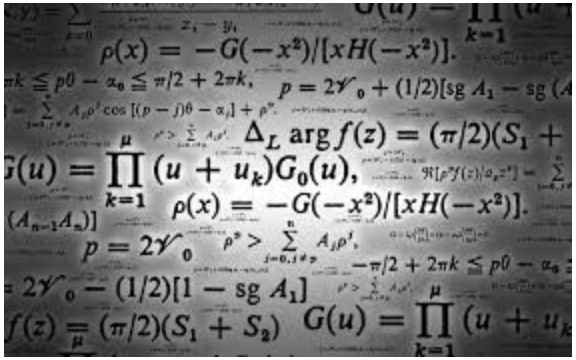


Computers are Machines that Execute Algorithms

- Two things computers do:
  - Performs simple **operations**

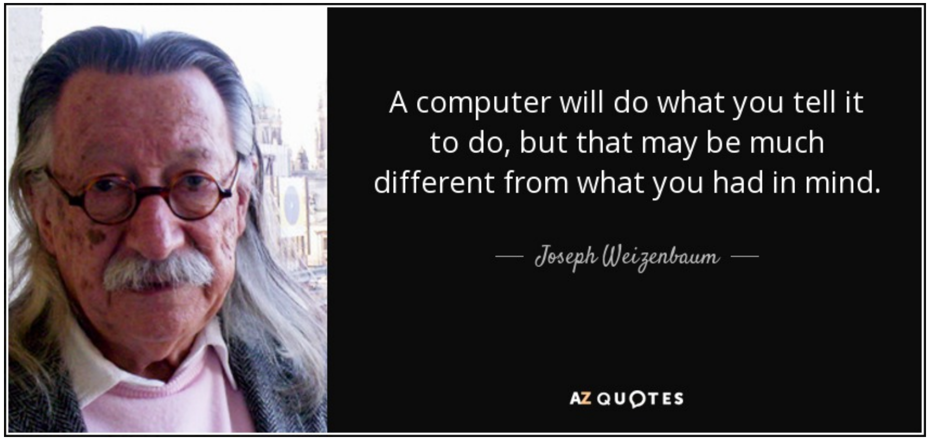
    100s of billions per second!
  - **Remembers** results

    100s of gigabytes of storage!



- What kinds of calculations?
  - **Built-in** to the machine, e.g., +
  - Ones that **you define** as the programmer
- A computer will do what **you tell** it to do

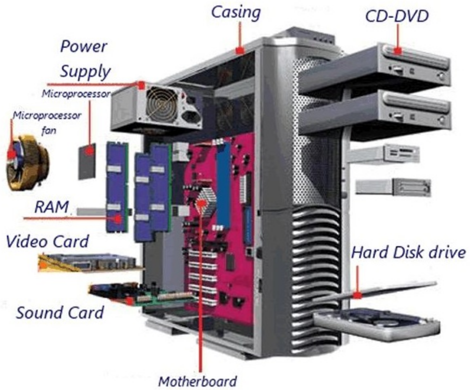
Don’t Blame the Machine

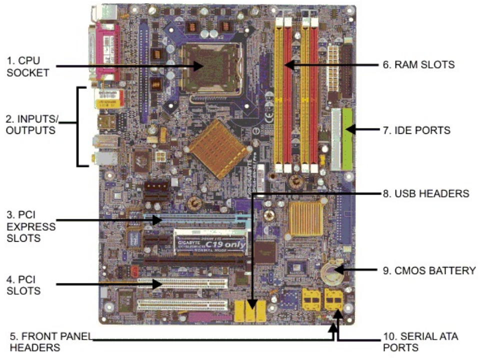


"It only does what you tell it to do" #programmer

Computers Are Machines that Execute Algorithms

- **Fixed program** computer
  - Fixed set of algorithms
  - What we had until 1940's
- **Stored program** computer
  - Machine stores and executes instructions
- **Key insight:** Programs are no different from other kinds of data

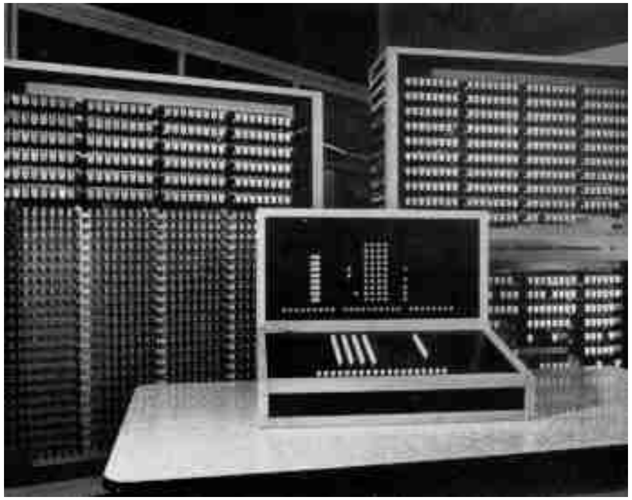




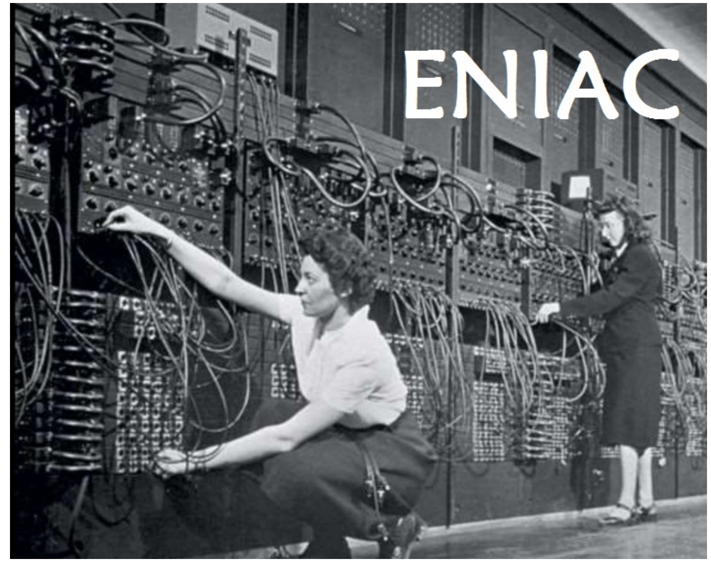
STORED PROGRAM COMPUTER

- Sequence of **instructions stored** inside computer
  - Built from predefined set of primitive instructions
    1) Arithmetic and logical
    2) Simple tests
    3) Moving data
- Special program (interpreter) **executes each instruction in order**
  - Use tests to change flow of control through sequence
  - Stops when it runs out of instructions or executes a halt instruction

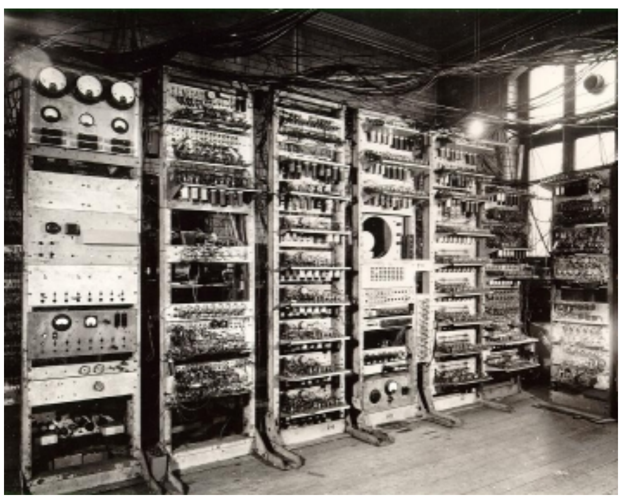
A Short History of Programmable Computers



Konrad Zuse's Z3, 1941 64 bytes



ENIAC, 1945, 200 bytes

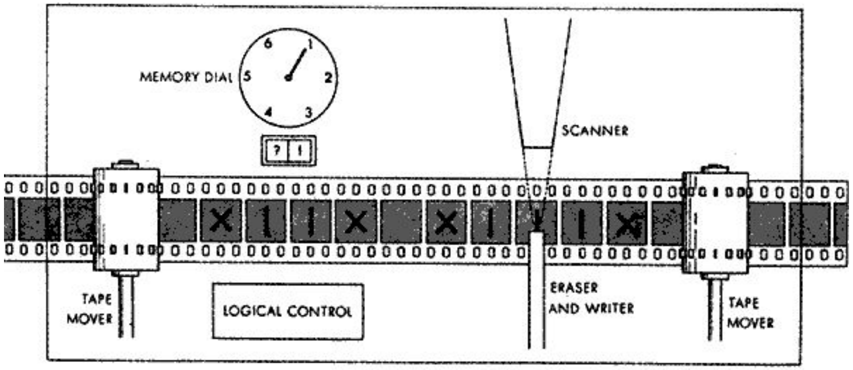


SSEM, 1948, 1024 bytes, 0.0011 MIPS
(first to put data and code in same memory)

iPhone 7, 2017, 3G bytes, 3,500 MIPS

BASIC PRIMITIVES

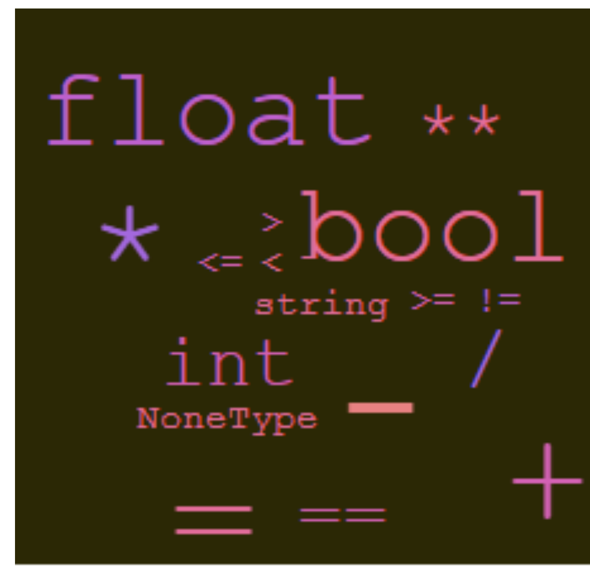
- Turing showed that you can **compute anything** with a very simple machine with only 5 primitives



- Real programming languages have
  - More convenient set of primitives
  - Ways to combine primitives to **create new primitives**
- Anything computable in one language is computable in any other programming language
  - It’s about convenience, not power

ASPECTS OF LANGUAGES

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



ASPECTS OF LANGUAGES

- **syntax**
  - English: "`cat dog boy`" → not syntactically valid

    "`cat hugs boy`" → syntactically valid
  - programming language: `"hi"5` → not syntactically valid

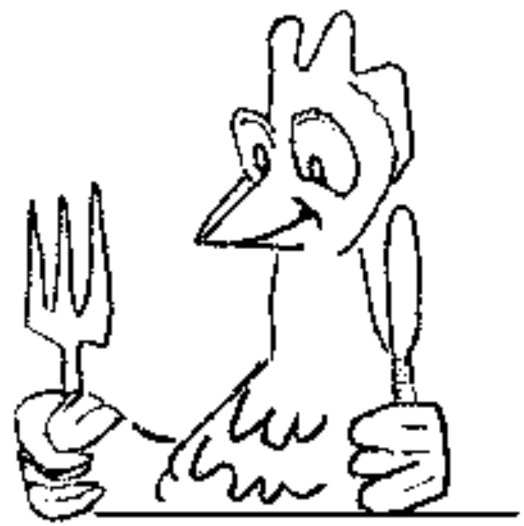
    `"hi"*5` → syntactically valid

ASPECTS OF LANGUAGES

- **Static semantics**: which syntactically valid strings have meaning
  - English: `"I are hungry"` → syntactically valid
    but static semantic error
  - PL: `"hi"+5` → syntactically valid
    but static semantic error

ASPECTS OF LANGUAGES

- **Semantics**: the meaning associated with a syntactically correct string of symbols with no static semantic errors
- English: can have many meanings `”The chicken is ready to eat."`
- Programs: have only one meaning
- But may not be what programmer intended



WHERE THINGS GO WRONG

- **Syntactic errors**
  - Common and easily caught
- **Static semantic errors**
  - Some languages check for these before running program
  - Can cause unpredictable behavior
- No linguistic errors, but **different meaning than what programmer intended**
  - Program crashes, stops running
  - Program runs forever
  - Program gives an answer, but it’s wrong!

PYTHON PROGRAMS

- A **program** is a sequence of definitions and commands
  - Definitions **evaluated**
  - Commands **executed** by Python interpreter in a shell
- **Commands** (statements) instruct interpreter to do something
- Can be typed directly in a **shell** or stored in a **file** that is read into the shell and evaluated
  - Problem Set 0 will introduce you to these in Anaconda

Five Minute Break

OBJECTS

- Programs manipulate **data objects**
- Objects have a **type** that defines the kinds of things programs can do to them
  - `30` is a number so we can add/sub/mult/div/exp/etc
  - `‘John’` is a string so we can look at substrings of it, but we can’t divide it by a number
- Objects can be
  - Scalar (cannot be subdivided)
  - Non-scalar (have internal structure that can be accessed)

SCALAR OBJECTS

- `int` – represent **integers**, ex. `5`
- `float` – represent **real numbers**, ex. `3.27`
- `bool` – represent **Boolean** values `True` and `False`
- `NoneType` – **special** and has one value, `None`
- can use `type()` to see the type of an object

```
>>> type(5)
int
>>> type(3.0)
float
```

what you write into the Python shell

what shows after hitting enter

TYPE CONVERSIONS (CAST)

- Can **convert object of one type to another**
  - `float(3)` converts the int `3` to float `3.0`
  - `int(3.9)` truncates the float `3.9` to int `3`
- Some operations perform implicit casts
  - `round(3.9)` returns the int `4`

EXPRESSIONS

- **Combine objects and operators** to form expressions
- An expression has a **value**, which has a type
- Syntax for a simple expression

  `<object> <operator> <object>`

OPERATORS ON ints and floats

- `i+j` → the **sum**
- `i-j` → the **difference**
- `i*j` → the **product**

if both are ints, result is int
if either or both are floats, result is float

- `i/j` → **division**

result is always a float

- `i//j` → **floor division**

What does it do?
What is type of output?

- `i%j` → the **remainder** when `i` is divided by `j`
- `i**j` → `i` to the **power** of `j`

SIMPLE OPERATIONS

- Parentheses used to tell Python to do these operations first
- **Operator precedence** without parentheses
  - **
  - * / % executed left to right, as appear in expression
  - + and – executed left to right, as appear in expression

BINDING VARIABLES AND VALUES

- Equal sign is an **assignment** of a value to a variable name
- An assignment binds a value to a name

variable value

```
pi = 355/113
```

- Compute the **right hand side** → **VALUE**
- Store it (bind it) to the **left hand side** → **VARIABLE**

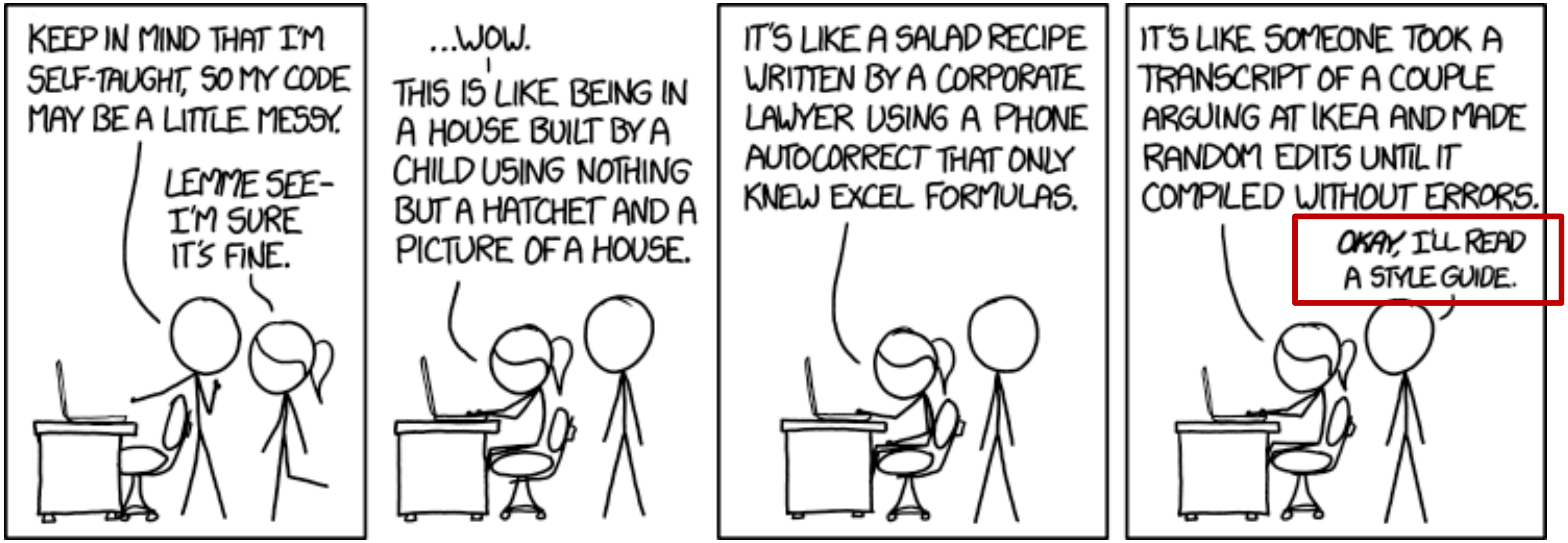
ABSTRACTING EXPRESSIONS

- Why **give names** to values of expressions?
- To **reuse names** instead of values
- Makes code easier to read and modify

```
#Compute approximate value for pi
pi = 355/113
radius = 2.2
area = pi*(radius**2)
circumference = pi*(radius*2)
```

- Choose variable names wisely
- Code needs to read
  - Today, tomorrow, next year
  - By author and others

Readability Matters

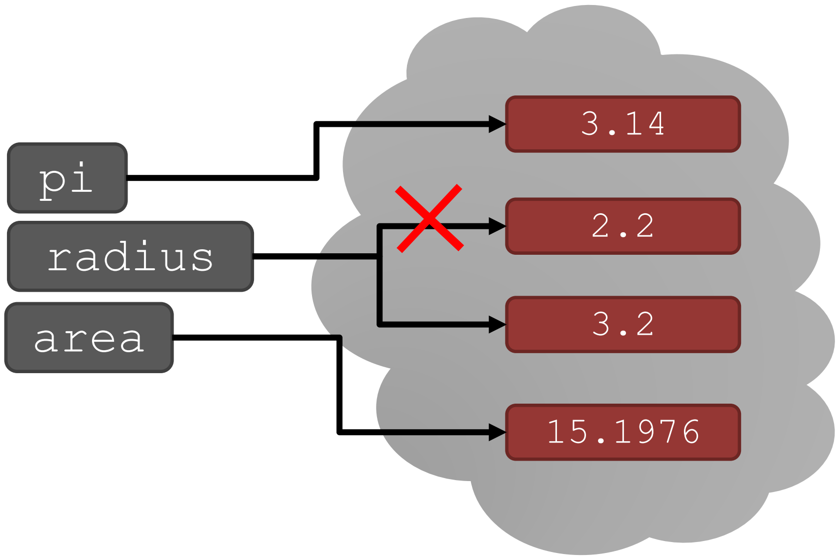


https://xkcd.com/1513/

CHANGING BINDINGS

- Can **re-bind** variable names using new assignment statements
- Previous value may still stored in memory but lost the handle for it

```
pi = 3.14
radius = 2.2
area = pi*(radius**2)
radius = radius+1
```



Second assignment to radius does not change value of area

BINDING EXAMPLE

- Swap values of x and y?

```
x = 1
y = 2
y = x
x = y
```

- Swap values of x and y?

```
x = 1
y = 2
temp = y
y = x
x = temp
```

How about this?

```
x = 1
y = 2
x,y = y,x
```

Right hand side of assignment is evaluated before any bindings are changed

STRINGS

- Letters, special characters, spaces, digits
- Think of an `str` as a **sequence** of case sensitive characters
- Enclose in **quotation marks or single quotes**

```
hi = "hello there"
```

- **Concatenate** strings

```
name = "John"
greeting = hi + " " + name
```

- Many other **operations** on strings
  - Hear all about them on Monday

Printing

- Used to **output** stuff to console
- Function is `print`

Note difference: print takes the values of objects, comma separated, and outputs each, space separated; or we can concatenate strings together, then print as single object

```
x = 1
print(x)
x_str = str(x)
print("my fav num is", x, ".", "x =", x)
print("my fav num is " + x_str + ". " + "x = " + x_str)
```

Note how in this case, we explicitly put in desired spaces

What about?

```
print = 3
print("Hello")
```

Input

- `x = input(s)`
    - prints the value of the string `s`
    - user types in something and hits enter
    - that value is assigned to the variable `x`
- binds that value to a variable

```
text = input("Type anything... ")
print(5*text)
```

- `input` always returns an **str,** must cast if working with numbers

```
num = int(input("Type a number... "))
print(5*num)
```

An Important Algorithm: Newton's Method

- Finds roots of a polynomial
  - E.g., find g such that $f(g, x) = g^3 - x = 0$
- Algorithm uses successive approximation, like Babylonian algorithm
- NextGuess = guess - $\dfrac{f(guess)}{f'(guess)}$

```
#Try Newton Raphson for cube root
print('Find the cube root of x')
x = 9
g = 3
print('Current estimate cubed =', g**3)
nextGuess = g - ((g**3 - x)/(3*g**2))
print('Next guess to try =', nextGuess)
```

Comparison Operators

- `i` and `j` are variable names
- Comparisons below evaluate to a **Boolean**

`i > j`

`i >= j`

`i < j`

`i <= j`

With strings, be careful about case sensitivity: 'March' != 'march', for example

`i == j` → **equality** test, `True` if `i` is the same as `j`

`i != j` → **inequality** test, `True` if `i` not the same as `j`

LOGICAL 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`

A	B	A and B	A or B
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

COMPARISON EXAMPLE

```
pset_time = 15
sleep_time = 8
print(sleep_time > pset_time)
drive = input('Are you planning to drive?')
drink = input('Are you sober?')
both = drink and drive
print(both)
```

But what good are they?

WHY bools?

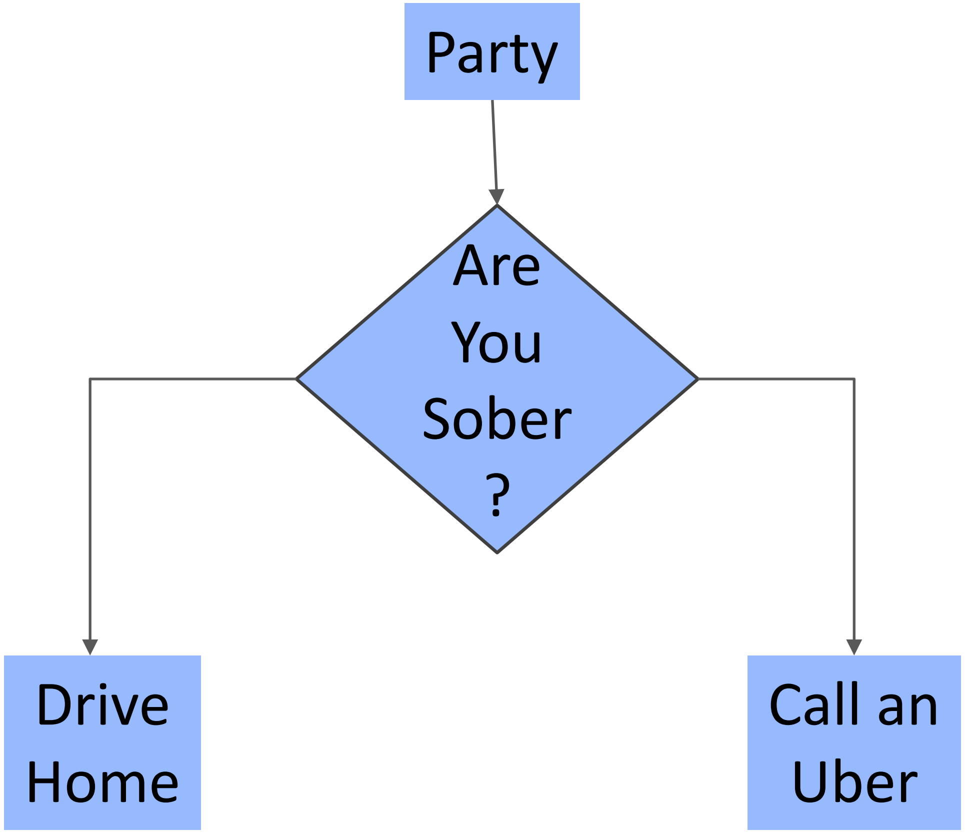
- When we get to flow of control, i.e. branching to different expressions based on values, we need a way of knowing if a condition is true
- E.g., if something is true, do this, otherwise do that

boolean

some commands

some commands

Because All Interesting Algorithms Involve Branching



CONTROL FLOW - BRANCHING

```
if <condition>:
    <statement>
    <statement>
    ...
```

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

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

- `<condition>` has a value `True` or `False`
- evaluate statements in that block if `<condition>` is `True`

INDENTATION MATTERS

- How you denote blocks of code

```
x = int(input("Enter a number for x: "))
y = int(input("Enter a different number for y: "))
if x == y:
    print("x and y are equal.")
    y = int(input("Enter a different number for y: "))
if x < y:
    print("x is smaller")
    if x < y/10 and x > y/100:
        print('x is an order of magnitute smaller')
    elif x < y/100:
        print('x is more than an order of magnitute smaller')
else:
    print("x is not smaller")
print("thanks!")
```

Semantically meaningful
Indentation is a good thing

Semantic structure
Should match
Visual structure

Monday

- Strings
- Iteration
- Some more useful algorithmic ideas