WELCOME!

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

6.0001 LECTURE 1

Ana Bell

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/fa18/6.0001
  - https://stellar.mit.edu/S/course/6/fa18/6.00
  - links to Piazza, MITx, Calendar, Grades, Psets, details on course policies
- **Problem sets website:** http://www.mit.edu/~6.00/
- Post privately on Piazza 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
    - Showing/sending code to others
  - Provide names of all “collaborators” on submission
  - We will be running a code similarity program on all psets
- **Extensions**
  - We consider extensions only with **S^3 support**
  - **Late days**, 3 to use per half semester

Grading, Problem Sets and Finger Exercises

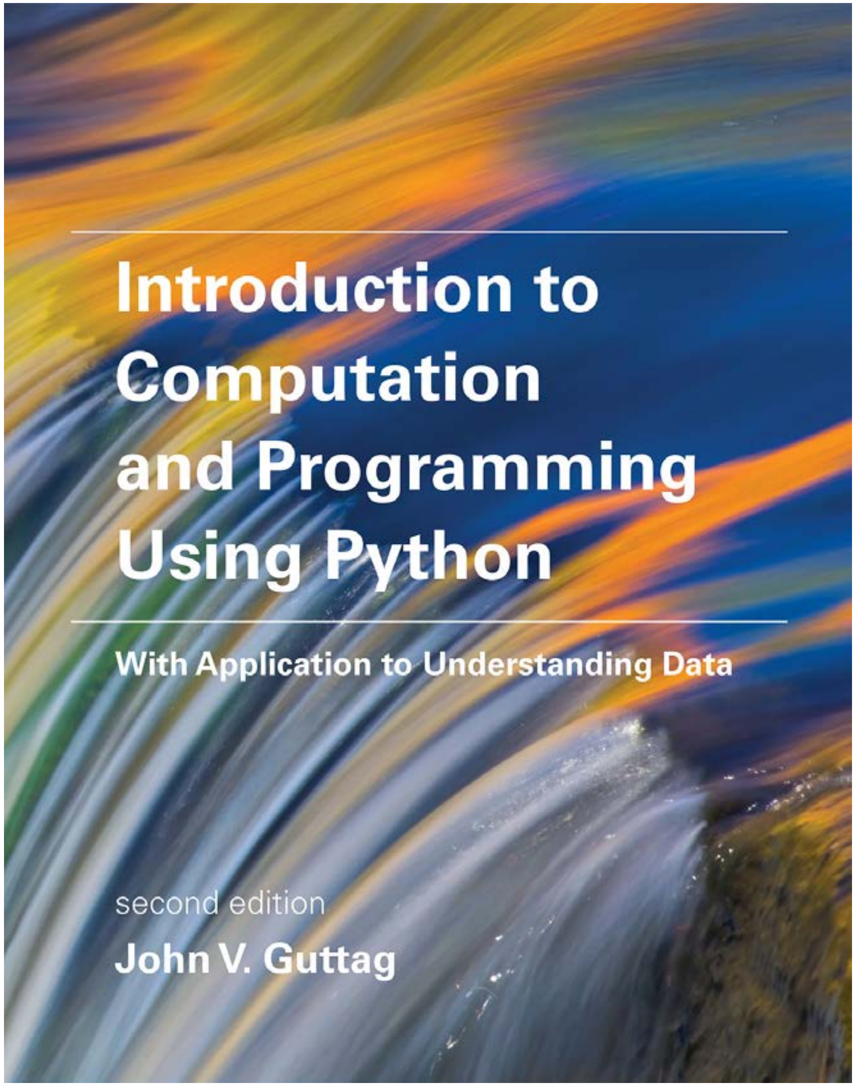
- **Problem sets**
  - 30% of final grade
  - 6.0001: 5 problem sets
  - 6.00: 10 problem sets
- **Finger exercises on MITx**
  - 10% of final grade for mandatory finger exercises
  - One for each lecture, due by the beginning of the next lecture

Grading, Exams and Quizzes

- **Microquizzes**
  - During class, in the **last 20 mins of some lectures** (see calendar)
  - Must have computer with wireless connection
  - If you need special accommodations, contact us asap
  - **6.0001: 3 of them** (worth 20%, best 2 out of 3)
  - **6.00: 6 of them** (worth 20%, best 2 out of 3 in each half semester)
  - **No makeups!**
- **Midterm (in-class)**
  - 6.0001: none
  - 6.00: worth 20% on Oct 17 (see calendar)
- **Final (in-class)**
  - 6.0001: worth 40% on Oct 17 (see calendar)
  - 6.00: worth 20% on Dec 12 (see calendar)
- 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

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
  - 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
- Learning to program
  - Can't passively absorb **programming as a skill**
  - **Download code before lecture** and follow along
  - Do MITx optional finger exercises
  - **Get help early**
  - Piazza, office hours, HKN tutoring:
    https://hkn.scripts.mit.edu/tutoring/
  - **Optional recitations** Fridays 10am and 2pm
- 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**

LET'S GO!

TYPES OF KNOWLEDGE

- **Declarative knowledge** is **statements of fact**
  - Someone will win a prize before class ends
- **Imperative knowledge** is a **recipe** or “how-to”
  1) Students sign up at `http://bit.ly/60001-raffle-fall18`
  2) Ana opens her IDE
  3) Ana chooses a random num between 1^{st} and n^{th} responder
  4) Ana finds the row number in the responders sheet.
  5) Locate the winner in class!
- Programming is about writing recipes to generate facts

WHAT YOU THINK YOU LOOK LIKE

WHAT YOU ACTUALLY LOOK LIKE

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

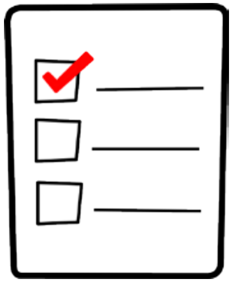
g	g*g	x/g	(g+x/g)/2
3	9	16/3	4.17
4.17	17.36	3.837	4.0035
4.0035	16.0277	3.997	4.000002

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**

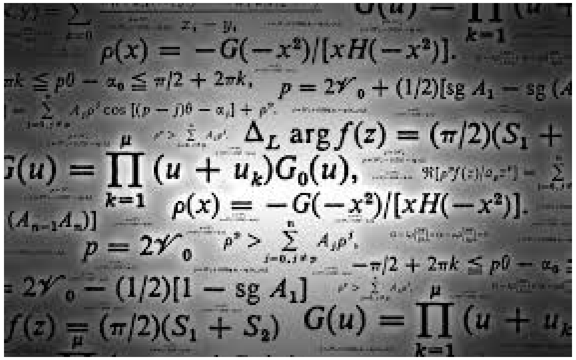


Computers are Machines that Execute Algorithms

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

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

    100s of gigabytes of storage!

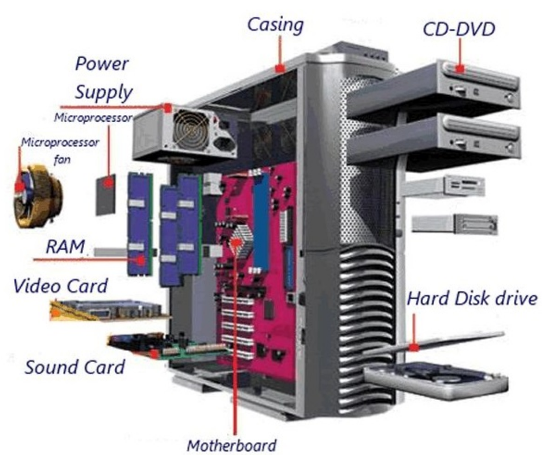


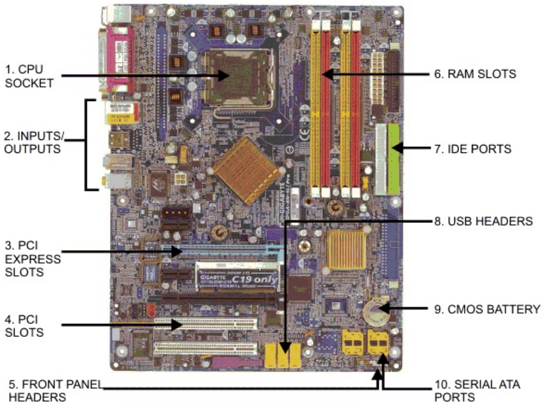
- What kinds of calculations?
  - **Built-in** to the machine, e.g., +
  - Ones that **you define** as the programmer
- The BIG IDEA here?

A computer will only do what you tell it to do #programmer #computerscience

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



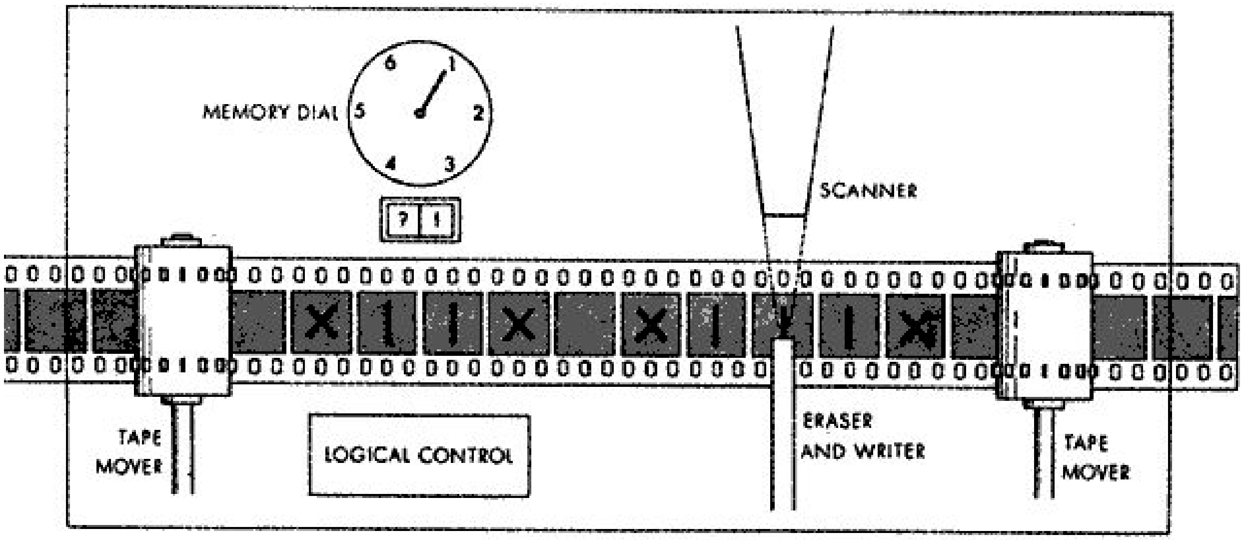


STORED PROGRAM COMPUTER

- 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

BASIC PRIMITIVES

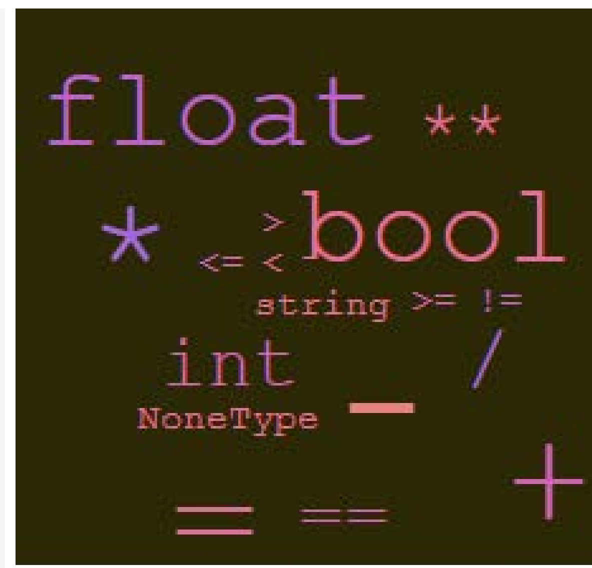
- Turing showed that you can **compute anything** with a very simple machine with only 6 primitives: left, right, print, scan, erase, no op



- 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

ASPECTS OF LANGUAGES

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



ASPECTS OF LANGUAGES

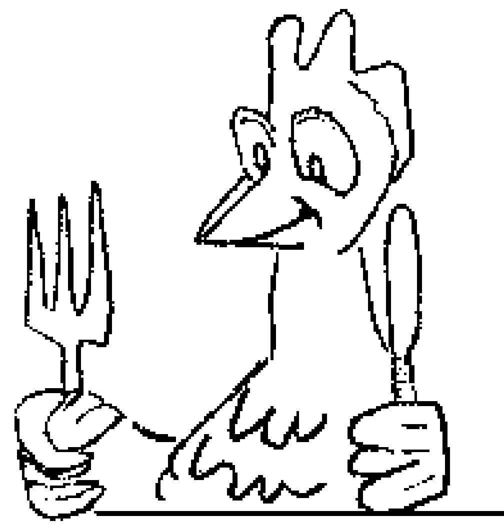
- **Syntax**
  - English: `"cat dog boy"` → not syntactically valid
    `"cat hugs boy"` → syntactically valid
  - programming language: `"hi"5` → not syntactically valid
    `"hi"*5` → syntactically valid

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 the meaning 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

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
  - `'Ana'` 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`

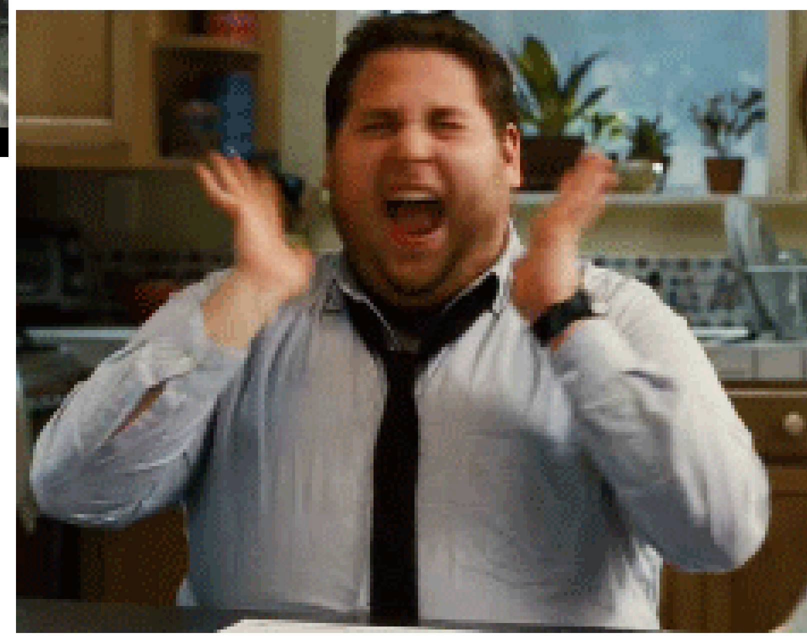
- `i**j` → `i` to the **power** of `j`

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

Five Minute Break

Trying to program using
only the 6 primitives



Installing Anaconda was successful

BINDING VARIABLES AND VALUES

LIVE EXERCISE

- Equal sign is an **assignment** of a value to a variable name
- Equal sign is not "solve for x"
- An assignment binds a value to a name

variable value

```
pi = 355/113
```

- Compute the value on the **right hand side** → **VALUE**
  - value stored in computer memory
- Store it (bind it) to the **left hand side** → **VARIABLE**
  - retrieve value associated with name or variable by invoking the name (typing it out)

ABSTRACTING EXPRESSIONS

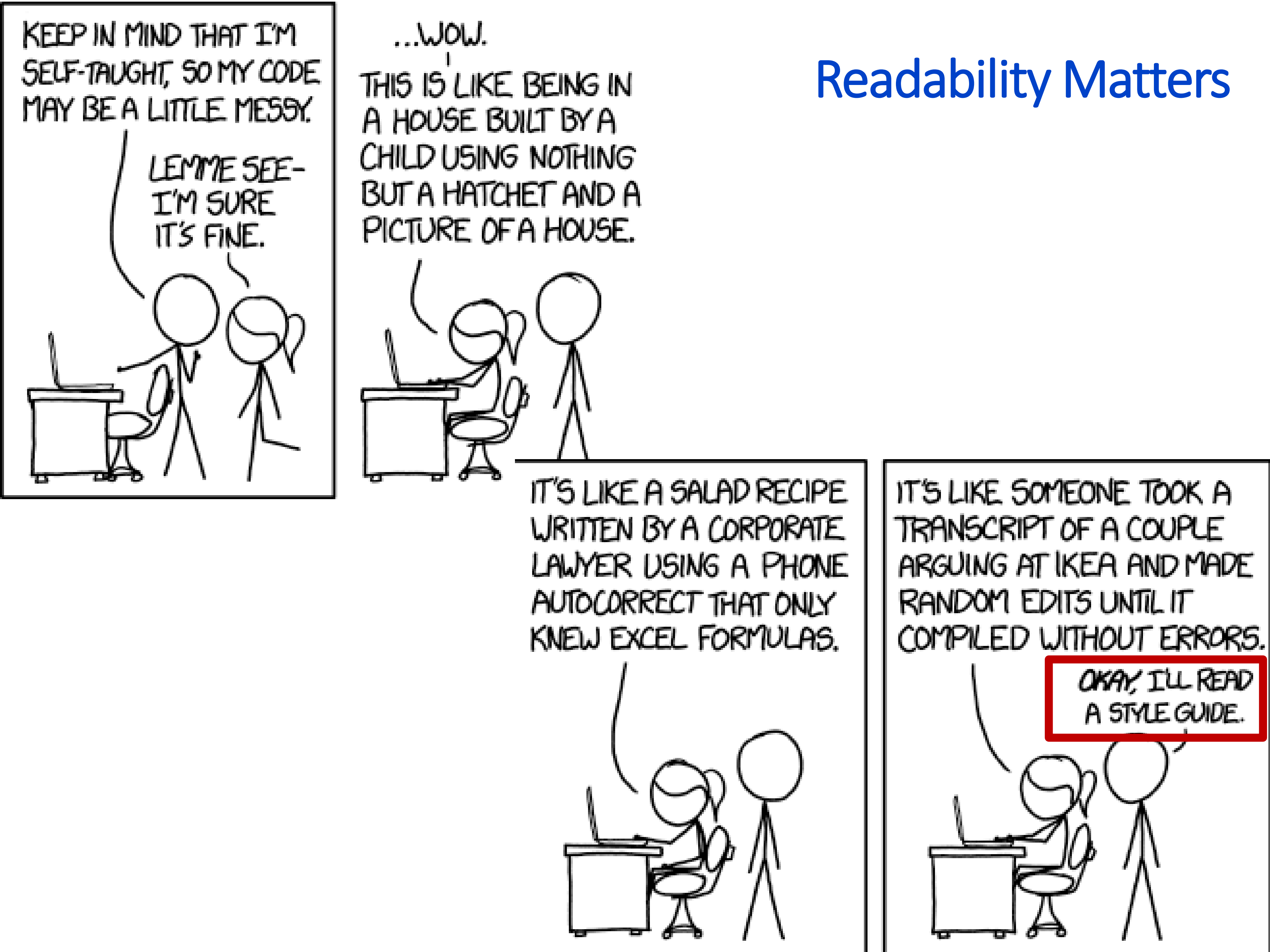
- Why **give names** to values of expressions?
  - To **reuse names** instead of values
  - Makes code easier to read and modify
- Choose variable names wisely
  - Code needs to read
    - Today, tomorrow, next year
    - By author and others

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

comments start with a # and are not part of code executed – used to tell others what your code is doing

an assignment
* expression on right
* variable name on left

Readability Matters

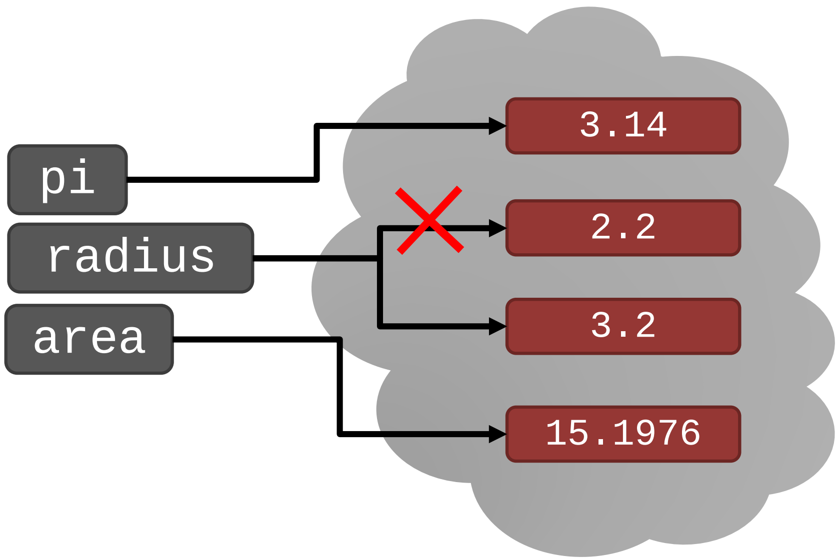


CHANGING BINDINGS

LIVE EXERCISE

- Can **re-bind** variable names using new assignment statements
- Previous value may still stored in memory but lost the handle for it
- Value for **area does not change** until you tell the computer to do the calculation again

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



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
```

STRINGS

LIVE EXERCISE

- 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 = "Ana"
greeting = hi + " " + name
```

- Do some **operations** on a string as defined in Python docs

```
silly = hi + " " + name * 3
```

- Many other **operations** on strings
  - Hear all about them next time

PRINTING

LIVE EXERCISE

- Used to **output** stuff to console

```
In [11]: 3+2
Out[11]: 5
```

"Out" tells you it's an interaction within the shell only

- Command is `print`

```
In [12]: print(3+2)
5
```

No "Out" means it is actually shown to a user, apparent when you edit/run files

- Printing many objects in the same command
  - Separate objects using commas, output them separated by spaces
  - Concatenate strings together, then print as single object

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

Put in spaces

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 - $\frac{f(guess)}{f'(guess)}$

```
#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`

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

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

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

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

LIVE EXERCISE

```
pset_time = 15
sleep_time = 8
print(sleep_time > pset_time)
derive = True
drink = False
both = drink and derive
print(both)
```

But what good are they?

WHY bools?

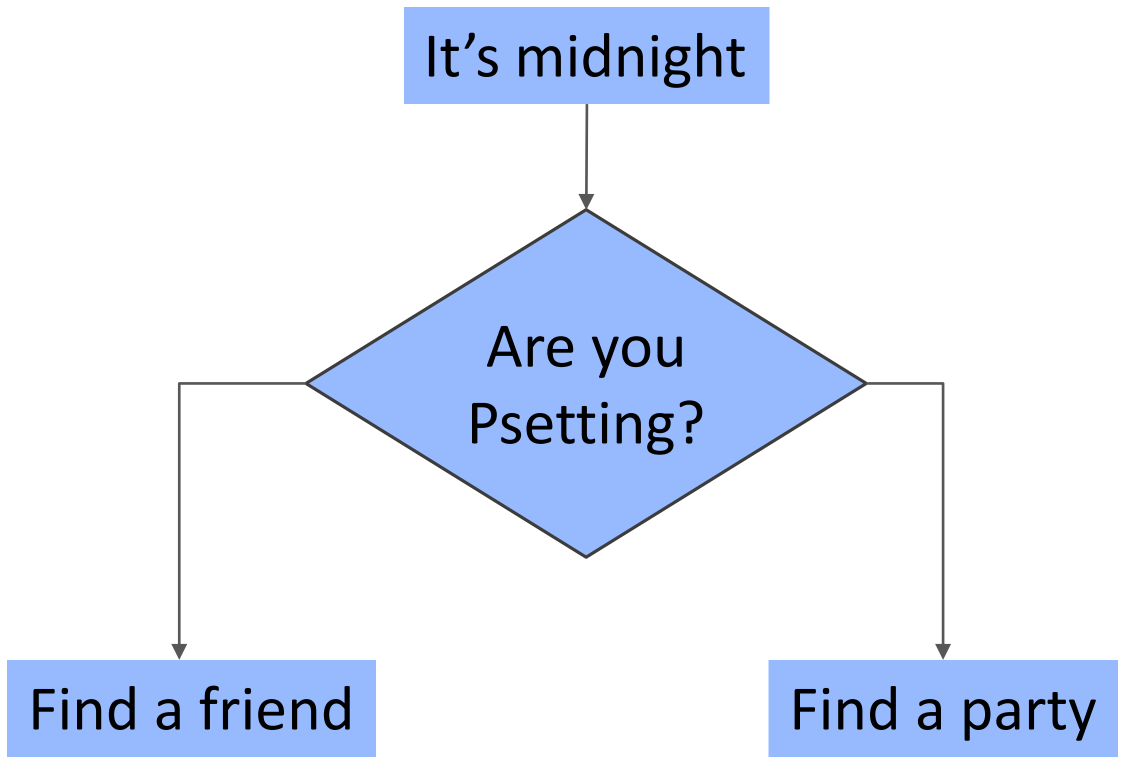
- When we get to flow of control, i.e. branching to different expressions based on values, we need a way of knowing if a condition is true
- E.g., if something is true, do this, otherwise do that

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

- **Semantic structure** matches **visual structure**

```
x = int(input("Enter a number for x: "))
y = int(input("Enter a different number for y: "))
if x == y:
    print(x,"and",y)
    print("These are equal!")
```

This code is correct

```
x = int(input("Enter a number for x: "))
y = int(input("Enter a different number for y: "))
if x == y:
    print(x,"and",y)
print("These are equal!")
```

This one is not correct due to bad indentation.

Monday

- More strings
- More branching
- Iteration
- Some more useful algorithmic ideas