

STRINGS, BRANCHING, ITERATION

(download slides and .py files to follow along!)

6.0001 LECTURE 2

Checklist

- Tablet works ✓
- Projector works ✓
- Slides are posted ✓
 - Links work
- Left is which way? Right is which way? ✓

LAST TIME

- Computers are dumb, execute step by step
- Syntax and semantics
- Scalar objects
- Simple operations
- Expressions, variables and values
- Storage and binding
- Input & output
- Branching and conditionals
- Indentation

Step by step execution

- Python tutor by Philip Guo
- <http://pythontutor.com/>
- Sqrt example

For sqrt fans

- Check Wikipedia

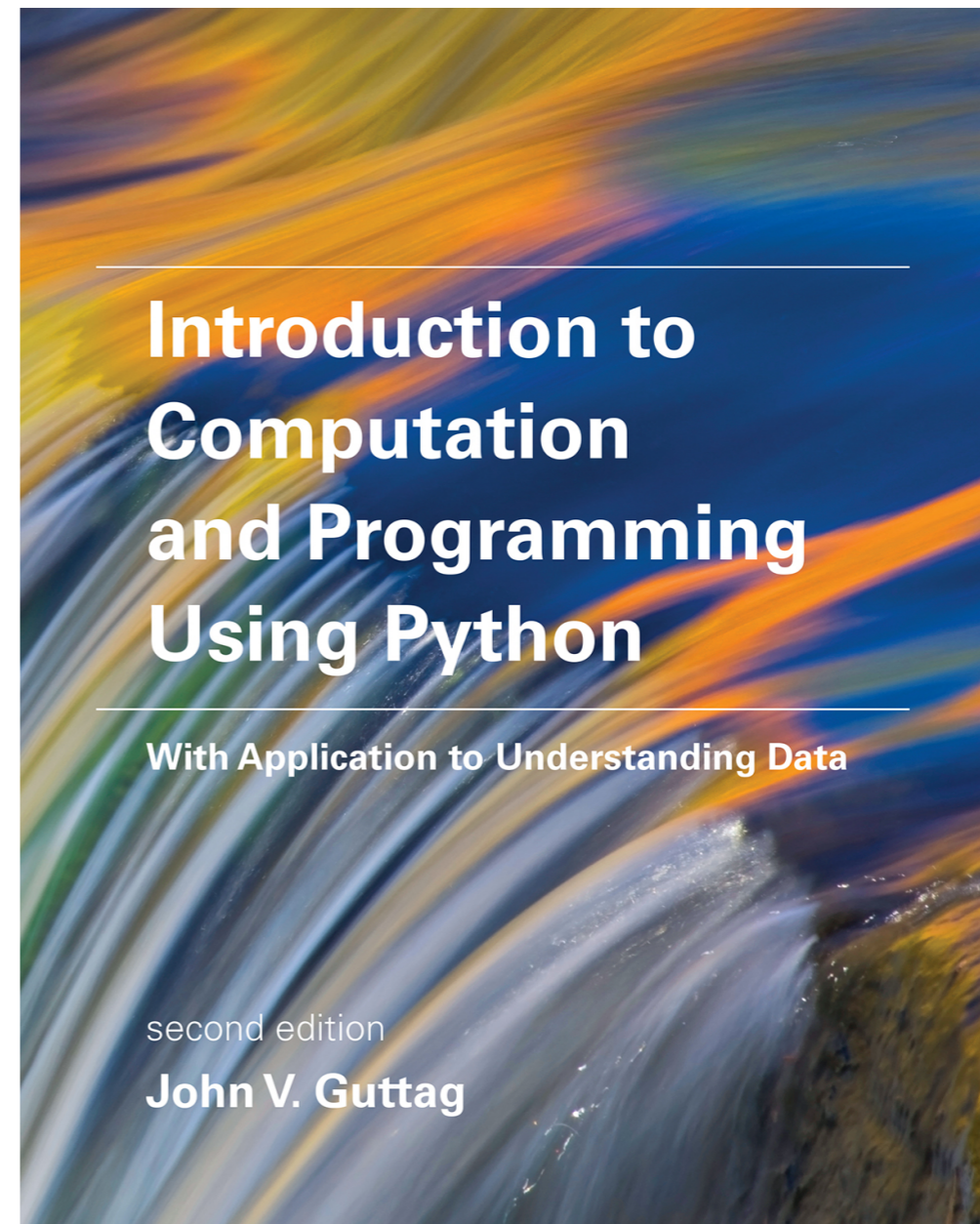
- https://en.wikipedia.org/wiki/Methods_of_computing_square_roots
- There is a whole industry of initial guesses
- Leverage scientific or binary representations
- Iterative methods

TODAY

- Recap of assignment, branching
- String object type
- Iteration and loops
- Guess-and-check algorithms

Assigned Reading

- ***Sections 2.3, 2.4***
- ***Sections 3.1, 3.2***



TYPES OF OBJECTS (RECAP)

- Variables and expressions
 - `int`
 - `float`
 - `bool`
 - `NoneType` ← **New**
 - `string` ← **New**
 - ... and others we will see later

VARIABLES (RECAP)

- Need a way to refer to computed values abstractly – give them a “name”
- **name**
 - descriptive
 - meaningful
 - helps you re-read code
 - should not be keywords
- **value**
 - information stored
 - can be updated

STRINGS (RECAP)

- Made up from letters, special characters, spaces, digits
- Think of as a **sequence** of case sensitive characters
- Enclose in **quotation marks or single quotes**
`today = 'Monday'`
- **Concatenate** strings
`this = "it is"`
`what = this + today`
`what = this + " " + today`
- Do some **operations** on a string as defined in Python docs
`announce = "It's " + today * 3`

OPERATOR OVERLOAD

- **Same operator** used on **different object types**
- **+** operator
 - E.g. Between two numbers: adds
 - E.g. Between two strings: concatenates
- ***** operator
 - E.g. Between two numbers: multiplies
 - E.g. Between a number and a string: repeats the string

STRING OPERATIONS

- Can compare strings with `==`, `>`, `<` etc.
 - compares letters one by one, order is alphabetical
 - E.g. `'a' < 'b'` return `True`, so does `'alex' < 'bea'`
 - Try it in console if you're not sure
- `len()` is a function used to retrieve the **length** of the string in the parentheses

```
s = "abc"
```

```
len(s) → evaluates to 3
```

STRINGS

- Square brackets used to perform **indexing** into a string to get the value at a certain index/position

```
s = "abc"
```

index 0 1 2
-3 -2 -1

```
s[0] → 'a'  
s[1] → 'b'  
s[2] → 'c'  
s[3] error  
s[-1] c  
s[-2] b  
s[-3] a
```

STRINGS

- Square brackets used to perform **indexing** into a string to get the value at a certain index/position

`s = "abc"`

index: 0 1 2 ← indexing always starts at 0

index: -3 -2 -1 ← last element always at index -1

`s[0]` → evaluates to "a"

`s[1]` → evaluates to "b"

`s[2]` → evaluates to "c"

`s[3]` → trying to index out of bounds, error

`s[-1]` → evaluates to "c"

`s[-2]` → evaluates to "b"

`s[-3]` → evaluates to "a"

'10'

s[1]

→

'0'

STRINGS

- Can **slice** strings using `[start:stop:step]`
- If give two numbers, `[start:stop]`, **step=1** by default
- Get characters at start until stop-1
- You can also omit numbers and leave just colons



SLICING STRINGS EXAMPLE

- Recall: `s[start:stop:step]`

`s = "abcdefgh"`
0 1 2 3 4 5 6 7

`s[3:6]`

`s[3:6:2]`

`s[3::]`

`s[::-1]`

`s[4:1:-2]`

→ def
df
defgh
hgfedcba

SLICING STRINGS EXAMPLE

`s = "abcdefgh"`

index: 0 1 2 3 4 5 6 7
index: -8 -7 -6 -5 -4 -3 -2 -1

*If unsure what some
command does, try it
out in your console!*

`s[3:6]` \rightarrow evaluates to "def", same as `s[3:6:1]`

`s[3:6:2]` \rightarrow evaluates to "df"

`s[3::]` \rightarrow evaluates to "defgh", same as `s[0:len(s):1]`

`s[::-1]` \rightarrow evaluates to "hgfedcba", same as `s[-1:-len(s):-1]`

`s[4:1:-2]` \rightarrow evaluates to "ec"

STRINGS

- Strings are “**immutable**” – cannot be modified

```
s = "car"
```

I could try s[0] = 'b'
→ error

STRINGS

- Strings are “**immutable**” – cannot be modified (but variables can be re-bound)

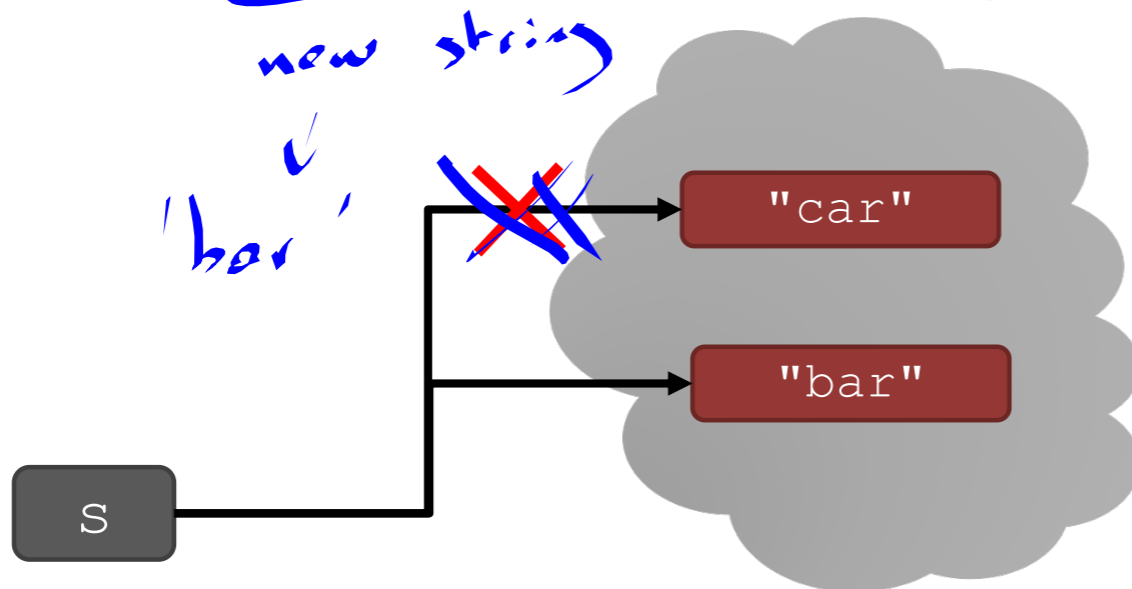
```
s = "car"
```

```
s[0] = 'b'
```

```
s = 'b'+s[1:len(s)]
```

→ gives an error

→ is allowed,
s bound to new object

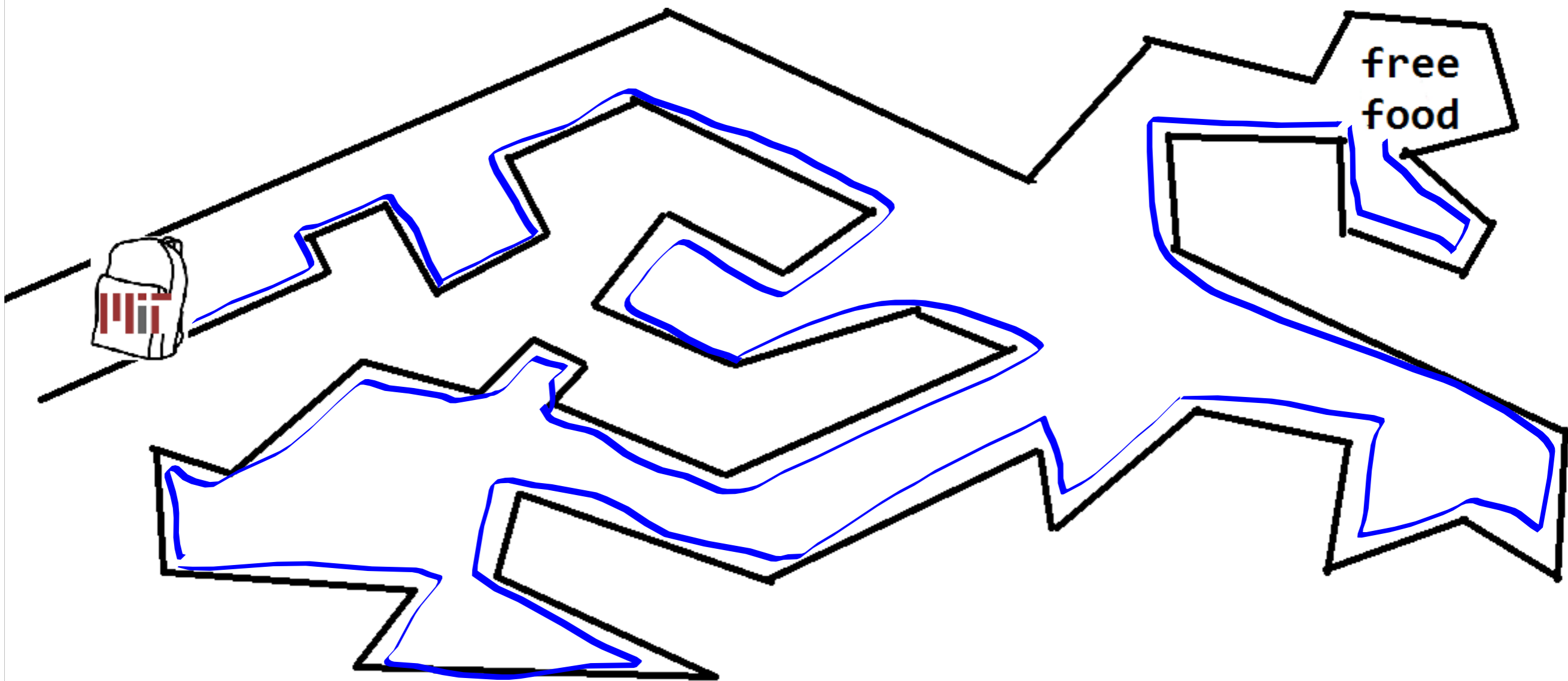


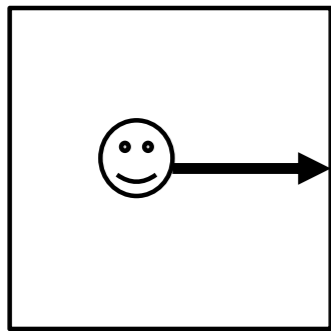
Strings recap

- Sequence of characters
- Access and slice with []
- Standard operations: +, *, ==, >, <
- Immutable

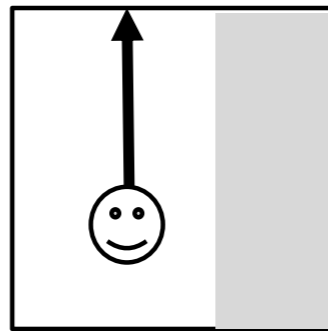
BOOLS (RECAP)

- Boolean values
 - `True`
 - `False`
- Useful with conditions
 - In **branching**:
If it's hot, go to the beach, otherwise stay at home.
 - In **repetitions**
As long as it's sunny, keep eating ice cream.

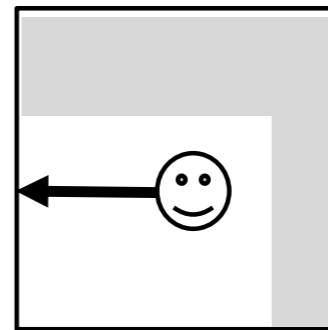




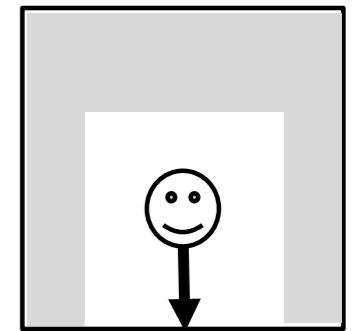
If right clear,
go right



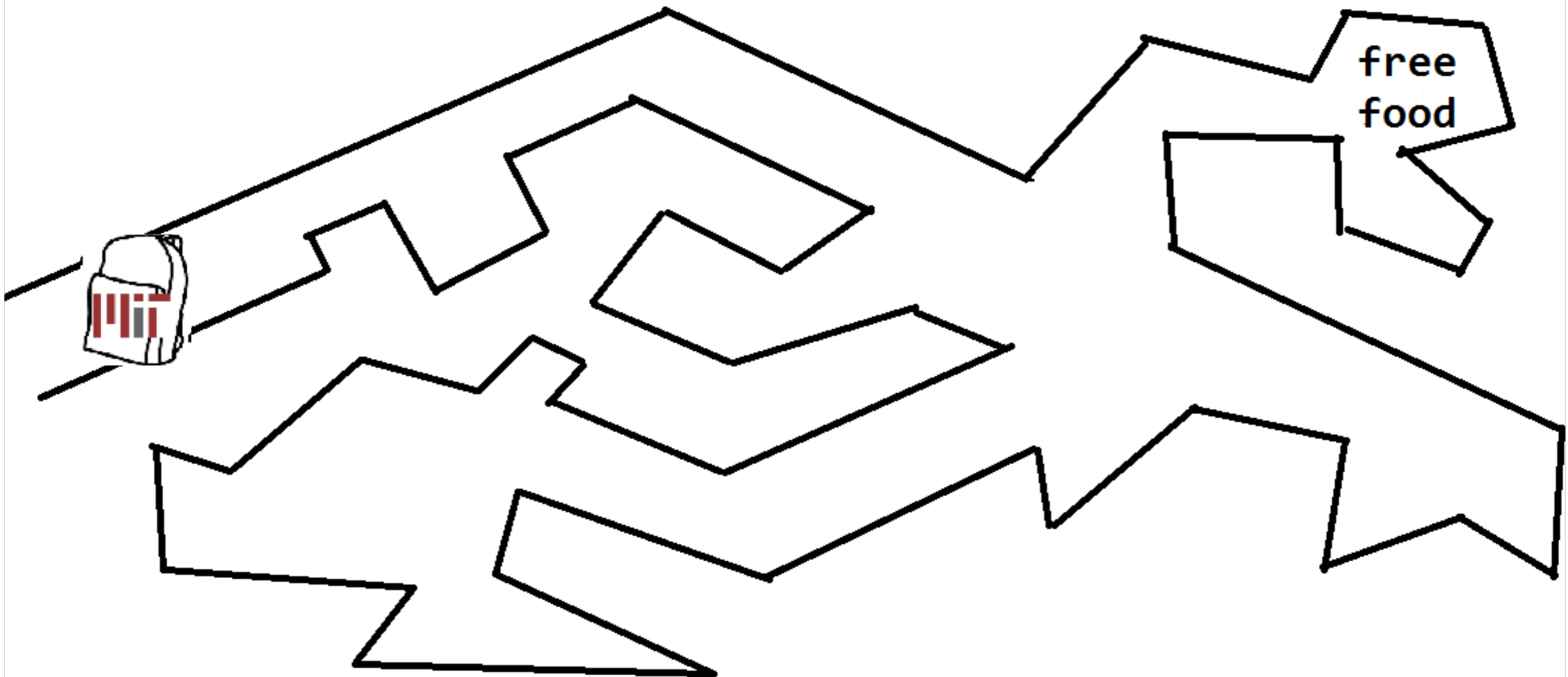
If right blocked,
go forward



If right and
front blocked,
go left



If right , front,
left blocked,
go back



BRANCHING

```
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 and BLOCKS

- Matters in Python

- How you **denote blocks of code**

```
x = int(input("Enter a number for x: "))
y = int(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!")
```

Handwritten annotations:

- Red circle around `==` in the `if` statement.
- Red "5" next to the first `input` line.
- Red "5" next to the second `input` line.
- Red "True" next to the `if x == y:` line.
- Red "True" next to the `if y != 0:` line.
- Red "1.0" next to the `print("therefore, x / y is", x/y)` line.
- Red "executes" with an arrow pointing to the `print("thanks!")` line.



INDENTATION and BLOCKS

- Matters in Python
- How you **denote blocks of code**

```
x = int(input("Enter a number for x: "))
y = int(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!")
```

5 5 0 0
5 0 0 5
False

True
✓

✓

```

x = int(input("Enter a number for x: "))
y = int(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!")

```

only applies to
if/elif/else

~~elif~~ ^{if x != 0:} x < y:

would be the
same here

would incorrectly
execute when

$x == y$

if $x < 0$ or $y < 0$:
and

INDENTATION and BLOCKS

- Matters in Python
- How you **denote blocks of code**

```
x = int(input("Enter a number for x: "))
y = int(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!")
```

5	5	0
5	0	0
True	False	True
<-		<-
True		False
<-		
	False	
	<-	
<-	<-	<-

CONTROL FLOW: while LOOPS

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

while True:
 print '!'

- <condition> **evaluates to a Boolean**
- If <condition> is True, **execute all the steps inside** the while code block
- **Check** <condition> again
- **Repeat** until <condition> is False
- If <condition> is never False, then will loop forever!!

Recall square root

```
x = 16.0
```

```
g = 3.0
```

```
tolerance = 0.0001
```

```
while abs( g * g - x ) > tolerance:  
    g = (g + x / g) / 2.0  
    print (g)
```



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

while LOOP EXAMPLE

You are in the Lost Forest.



Go left or right?

PROGRAM:

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

CONTROL FLOW: when `while` loops aren't ideal

- Iterate and print through numbers in a sequence, e.g. 0 to 4 (included)

a little complicated with `while` loop

`x = 0`

`while x < 5:`

`print(x)`

`x = x + 1`

`for x in range(5):`
 `print(x)`

CONTROL FLOW:

`while` vs. `for` LOOPS

- Iterate through numbers in a sequence

more complicated with while loop

```
n = 0
```

```
while n < 5:
```

```
    print(n)
```

```
    n = n+1
```

Set loop variable outside while loop

Test loop variable in condition

Increment loop variable inside while loop

$n = n+1$ equivalent to $n += 1$

shortcut with for loop

```
for n in range(5):
```

```
    print(n)
```

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 the **prev value + 1**
- etc. until `<variable>` gets **`some_num - 1`**

`range(start, stop, step)`

- Default values are `start = 0` and `step = 1` and optional
- Loop until value reaches `stop - 1`

```
mysum = 0
for i in range(7, 10):
    mysum += i
print(mysum)
```

```
mysum = 0
for i in range(5, 11, 2):
    mysum += i
print(mysum)
```

break STATEMENT

- Immediately exits whatever loop it is in
- Skips remaining expressions in code block
- **Exits only innermost loop!**

```
x = 16
for i in range(x):
    if i*i >= x:
        break
print(i)
```

for VS while LOOPS

for loops

- **know** number of iterations
- can **end early** via `break`
- uses a **counter**
- **can rewrite** a `for` loop using a `while` loop

while loops

- (potentially) **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

STRINGS AND LOOPS

```
s = "demo loops – fruit loops"
for index in range(len(s)):
    if s[index] == 'i' or s[index] == 'u':
        print("There is an i or u")
```

STRINGS AND LOOPS

```
s = "demo loops – fruit loops"
for index in range(len(s)):
    if s[index] == 'i' or s[index] == 'u':
        print("There is an i or u")
```

could be
`s[::2]`

```
for char in s:
```

```
    if char == 'i' or char == 'u':
        print("There is an i or u")
```

These two code snippets do the same thing; bottom one is more "pythonic"

variable char will be bound to each character of s in order

could be
s[::2]
= 'u':

These two code
snippets do the same
thing; bottom one is
more "pythonic"

to
in order

for char in s[::2]:

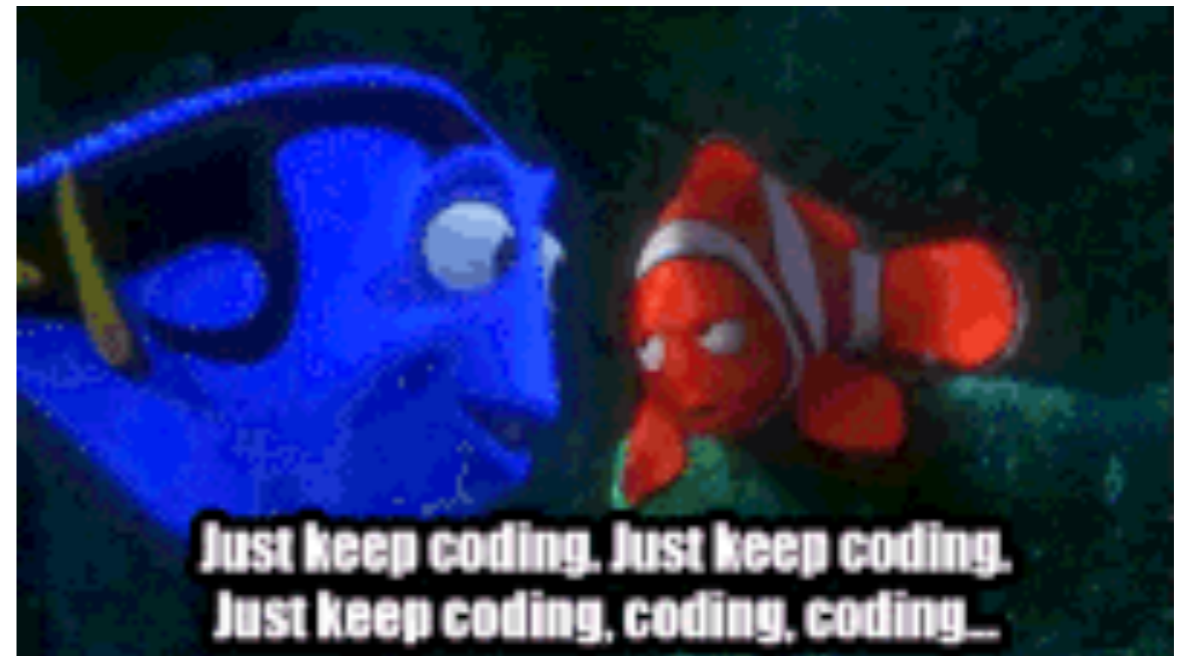
Recap: Control flow and blocks

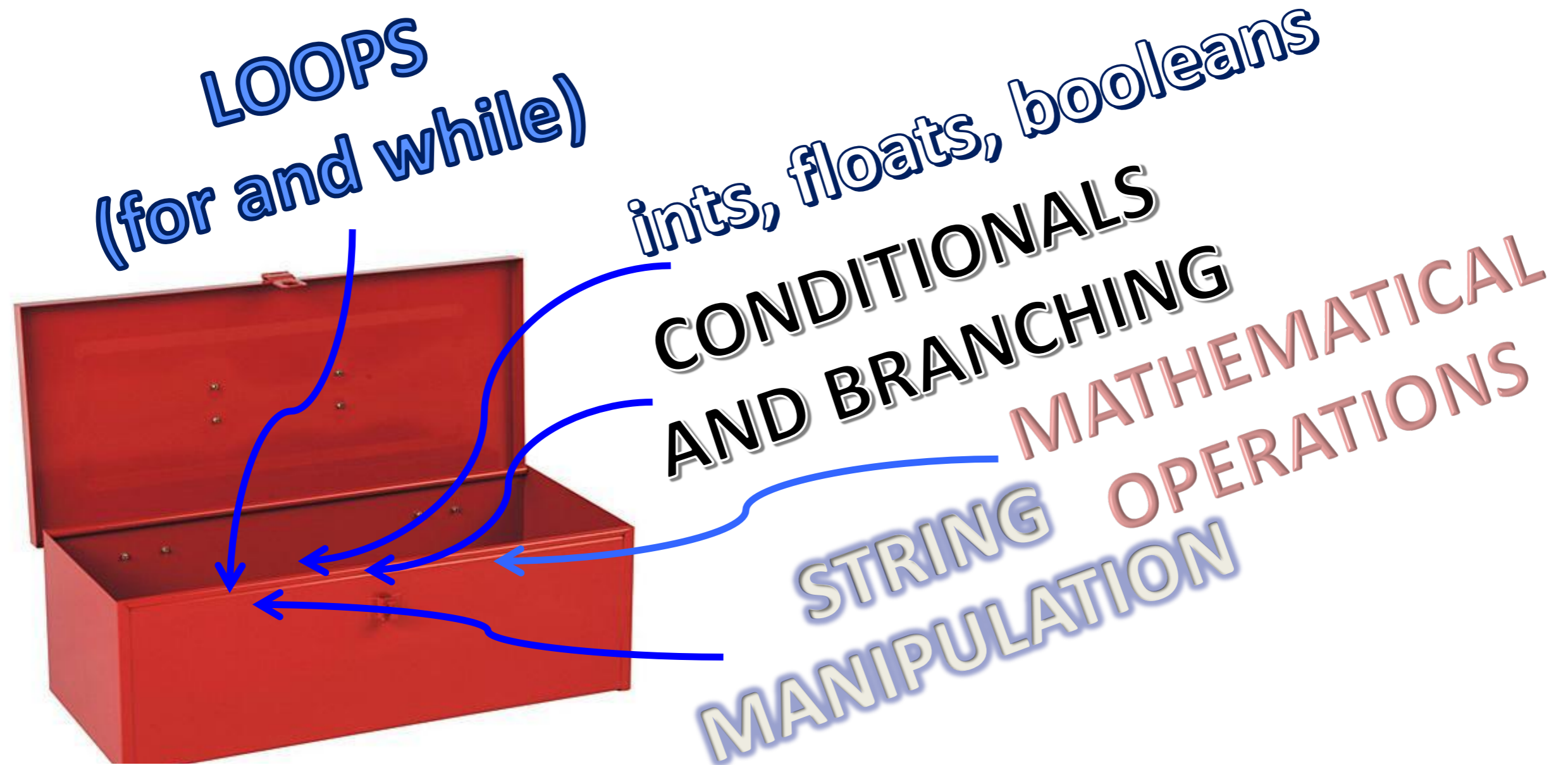
- If, else, elif
- while : unbounded, more general
- for
 - in range
 - in string
- (plus break if (really) needed)
- Block defined by indentation

Five Minute Break



Trying to fix my code





ALGORITHMS

GUESS-and-CHECK

BISECTION SEARCH

APPROXIMATION



GUESS-AND-CHECK

- Process called **exhaustive enumeration**
- Applies to a problem where ...
 - You are able to **guess a value** for solution
 - You are able to **check if the solution is correct**
 - You can **keep guessing** until
 - Find solution or
 - Have guessed all values

GUESS-AND-CHECK

– square root

- Basic idea:
 - Given an `int`, call it `x`, want to see if there is another `int` which is its square root
 - Start with a `guess` and check if it is the right answer
 - To be **systematic**, start with `guess = 0`, then 1, then 2, etc
- If `x` is a **perfect square**, we will **eventually find its root** and can stop
- But what if `x` is **not a perfect square**?
 - Need to know when to stop
 - **Use algebra** – if `guess squared` is bigger than `x`, then can stop

GUESS-AND-CHECK

– square root

```
guess = 0
x = int(input("Enter an integer: "))
while guess**2 < x:
    guess = guess + 1
if guess**2 == x:
    print("Square root of", x, "is", guess)
else:
    print(x, "is not a perfect square")
```

Exit loop when
 $guess**2 \geq x$

GUESS-AND-CHECK

– square root

- Does this work for any integer value of x ?
- What if x is negative?
 - `while` loop immediately terminates
 - Actually turns out to be correct
(if you don't believe in imaginary numbers)
- Could check for negative input, and handle differently

GUESS-AND-CHECK

– square root

```
guess = 0
neg_flag = False
x = int(input("Enter a positive integer: "))
if x < 0:
    neg_flag = True
while guess**2 < x:
    guess = guess + 1
if guess**2 == x:
    print("Square root of", x, "is", guess)
else:
    print(x, "is not a perfect square")
    if neg_flag:
        print("Just checking... did you mean", -x, "?")
```

while LOOP OR for LOOP?

- Already saw that code looks cleaner when iterating over sequence of values
 - Don't set up the iterant yourself as with a while loop
 - Less likely to introduce errors
- Consider an example that uses a `for` loop and an explicit `range` of values

GUESS-AND-CHECK

– cube root

```
cube = int(input("Enter an integer: "))
```

```
for guess in range(cube+1):
```

```
    if guess**3 == cube:
```

```
        print("Cube root of", cube, "is", guess)
```

GUESS-AND-CHECK

– cube root

```
cube = int(input("Enter an integer: "))
```

```
for guess in range(cube+1):
```

```
    if guess**3 == cube:
```

```
        print("Cube root of", cube, "is", guess)
```

*This ensures we get to
value of cube*

GUESS-AND-CHECK

– cube root

```
cube = int(input("Enter an integer: "))

for guess in range(abs(cube)+1):
    if guess**3 == abs(cube):
        if cube < 0:
            guess = -guess
        print("Cube root of "+str(cube)+" is "+str(guess))
```

GUESS-AND-CHECK

– cube root

```
cube = int(input("Enter an integer: "))
```

```
for guess in range(abs(cube)+1):
```

```
    if guess**3 == abs(cube):
```

```
        if cube < 0:
```

```
            guess = -guess
```

```
print("Cube root of " + str(cube) + " is " + str(guess))
```

*This properly handles
cube roots of negative
integers*

GUESS-AND-CHECK

– cube root

```
cube = int(input("Enter an integer: "))
```

```
for guess in range(abs(cube)+1):
```

```
    if guess**3 >= abs(cube):  
        break
```

```
if guess**3 != abs(cube):
```

```
    print(cube, "is not a perfect cube")
```

```
else:
```

```
    if cube < 0:
```

```
        guess = -guess
```

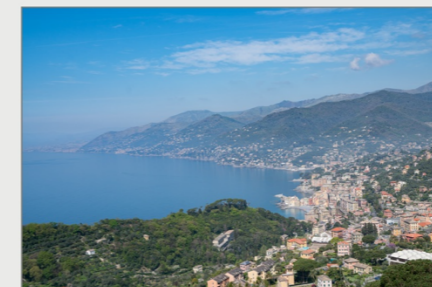
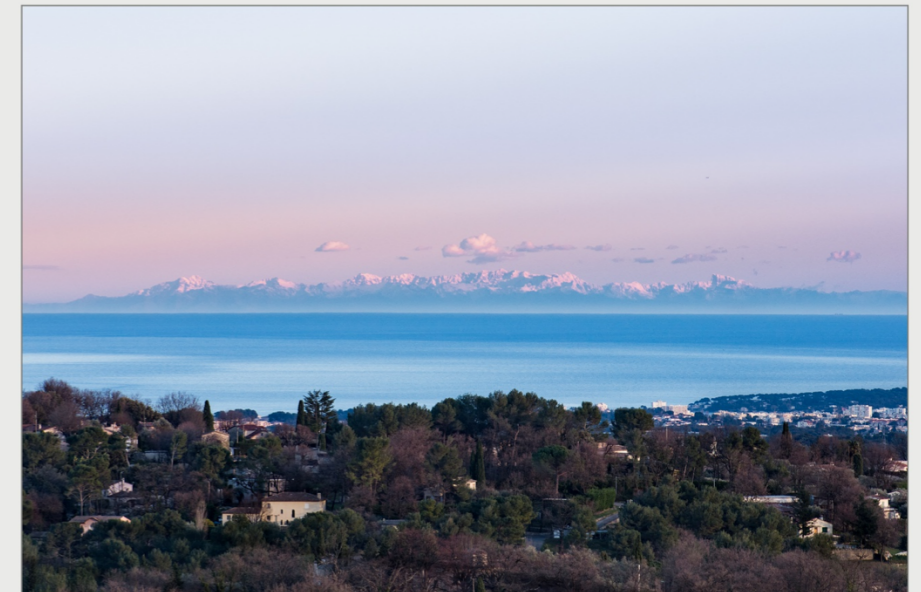
```
    print("Cube root of "+str(cube)+" is "+str(guess))
```

Terminate search once
know you have passed
possible answer

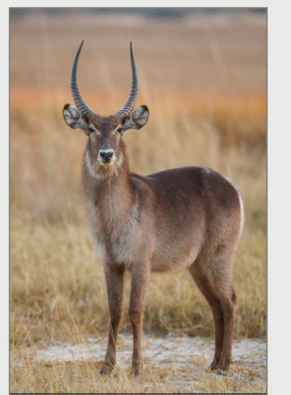
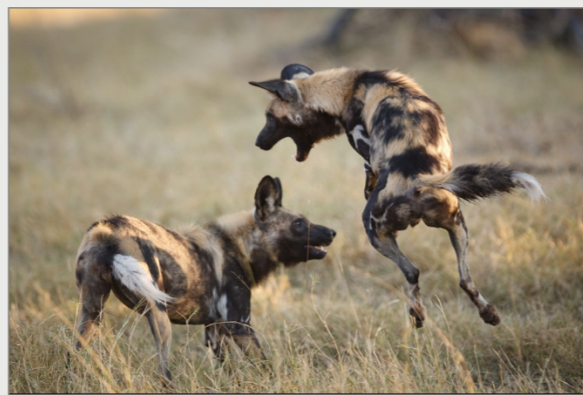
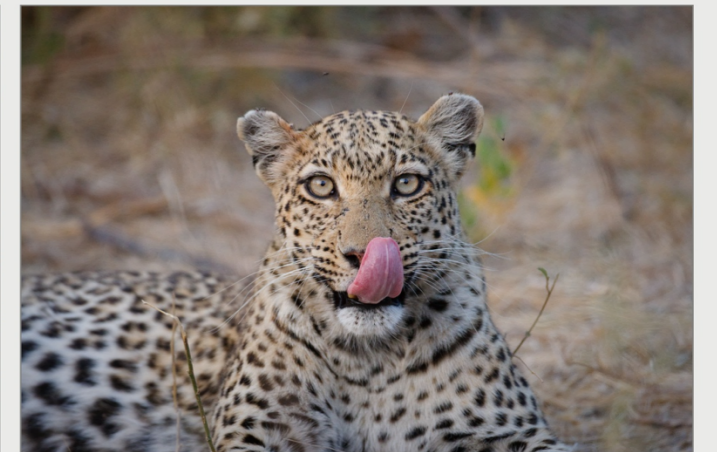
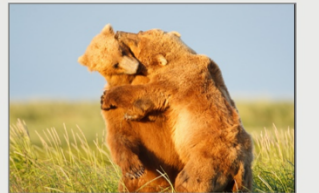
ANOTHER EXAMPLE

- Remember those word problems from your childhood?
- For example:
 - Alyssa, Ben, and Cindy are selling tickets to a fundraiser
 - Ben sells 20 fewer than Alyssa
 - Cindy sells twice as many as Alyssa
 - 1000 total tickets were sold by the three people
 - How many did Alyssa sell?
- Could solve this algebraically, but we can also use guess-and-check

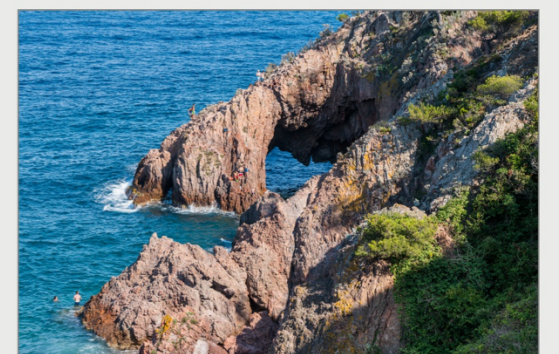
Fredo's guess and check



Fredo's guess and check



Fredo's guess and check



SUMMARY

- String data type
 - Strings can be indexed and sliced
 - Strings are immutable
- If elif else
- Looping mechanisms
 - `while` and `for` loops
 - Can loop over ranges of numbers
 - Can loop over elements of a string
- Indentation and blocks
- Exhaustive search (aka guess-and-check) provides a simple algorithm for solving problems where the set of potential solutions is enumerable