STRINGS, INPUT/OUTPUT, and BRANCHING

(download slides and .py files to follow along)

6.100L Lecture 2

Ana Bell

RECAP

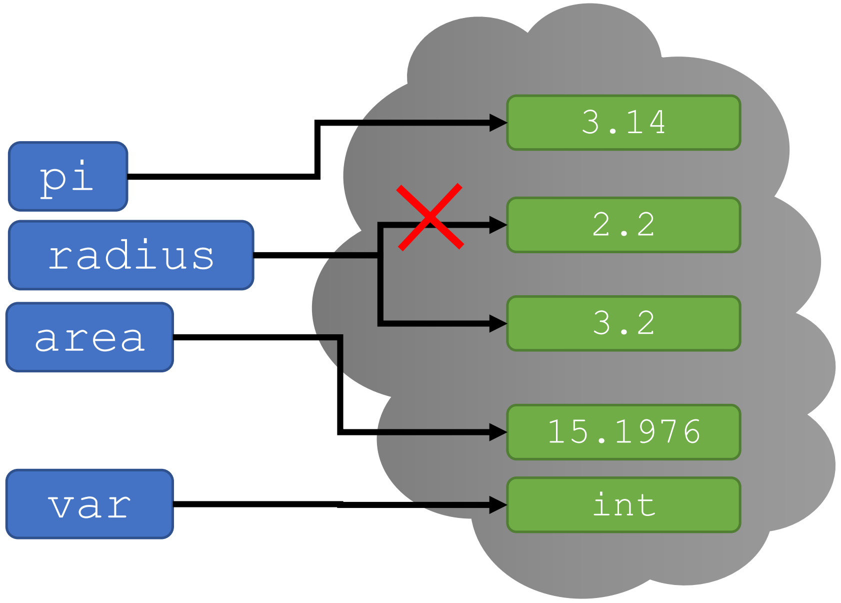
```
pi = 3.14

radius = 2.2

area = pi*(radius**2)

radius = radius+1


var = type(5*4)
```



- Objects
  - Objects in memory have **types**.
  - Types tell Python what **operations** you can do with the objects.
  - **Expressions evaluate to one value** and involve objects and operations.
  - Variables bind names to objects.
  - = sign is an assignment, for ex. `var = type(5*4)`

- Programs
  - Programs only **do what you tell them to do**.
  - Lines of code are executed **in order**.
  - Good variable names and comments help you **read code later**.

STRINGS

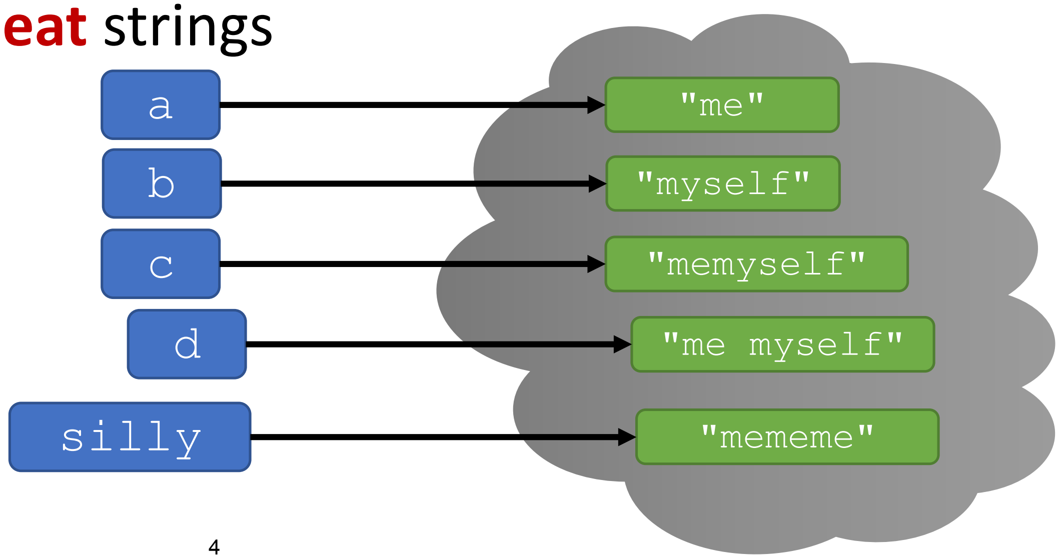
STRINGS

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

```
a = "me"
z = 'you'
```

- **Concatenate** and **repeat** strings

```
b = "myself"
c = a + b
d = a + " " + b
silly = a * 3
```



4

YOU TRY IT!

What's the value of s1 and s2?

-
  ```
  b = ":"
  c = ")"
  s1 = b + 2*c
  ```

-
  ```
  f = "a"
  g = " b"
  h = "3"
  s2 = (f+g)*int(h)
  ```

STRING OPERATIONS

- `len()` is a function used to retrieve the **length** of a string in the parentheses

```
s = "abc"
```

`len(s)` → evaluates to 3

`chars =` $\boxed{\texttt{len(s)}}$ Expression that evaluates to 3

SLICING to get ONE CHARACTER IN A STRING

- 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   ← index of last element is len(s) - 1 or -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"
```

SLICING to get a SUBSTRING

- Can **slice** strings using `[start:stop:step]`
- Get characters at **start**
  up to and including **stop-1**
  taking every **step** characters

This is confusing as you are starting out :(
Can't go wrong with explicitly giving start, stop, end every time.

- If give two numbers, `[start:stop]`, `step=1` by default
- If give one number, you are back to indexing for the character at one location (prev slide)
- You can also omit numbers and leave just colons (try this out!)

SLICING EXAMPLES

- Can **slice** strings using `[start:stop:step]`
- Look at step first. +ve means go left-to-right
  -ve means go right-to-left

```
s = "abcdefgh"
```

	a	b	c	d	e	f	g	h
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]` → evaluates to `"def"`, same as `s[3:6:1]`

`s[3:6:2]` → evaluates to `"df"`

`s[:]` → evaluates to `"abcdefgh"`, same as `s[0:len(s):1]`

`s[::-1]` → evaluates to `"hgfedcba"`

`s[4:1:-2]` → evaluates to `"ec"`

YOU TRY IT!

```
s = "ABC d3f ghi"

s[3:len(s)-1]
s[4:0:-1]
s[6:3]
```

IMMUTABLE STRINGS

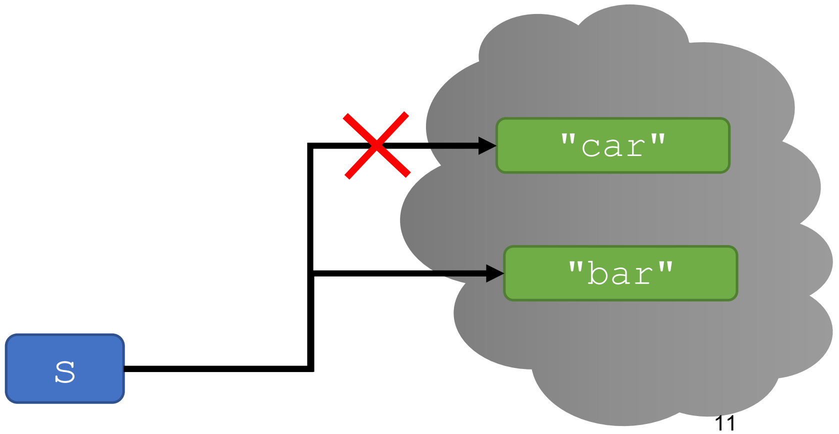
- Strings are "**immutable**" – cannot be modified
- You can create **new objects** that are versions of the original one
- Variable name can only be bound to one object

```
s = "car"

s[0] = 'b'
s = 'b'+s[1:len(s)]
```

→ gives an error

→ is allowed, s bound to new object



BIG IDEA

If you are wondering “what happens if”…

Just try it out in the console!

INPUT/OUTPUT

PRINTING

- 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 to output them separated by spaces
  - Concatenate strings together using + to print as single object

  -
    ```
    a = "the"
    b = 3
    c = "musketeers"
    print(a, b, c)
    print(a + str(b) + c)
    ```

Every piece being concatenated must be a string

14

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)
```

SHELL:

```
Type anything:
```

And it waits for characters and Enter to be hit

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**

“howdy”

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

SHELL:

```
Type anything: howdy
```

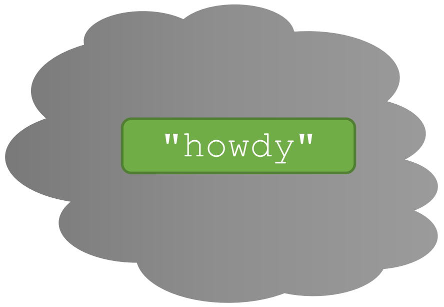
16

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)
```



SHELL:

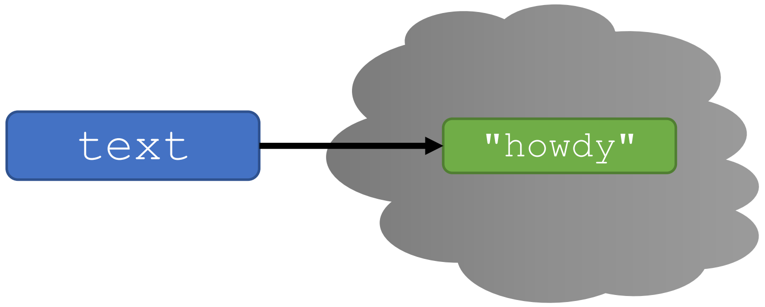
```
Type anything: howdy
```

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)
```



SHELL:

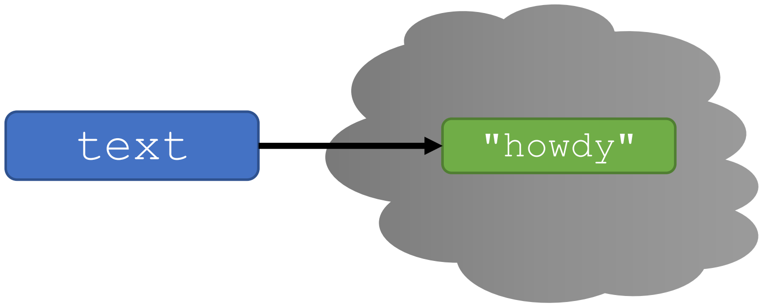
```
Type anything: howdy
```

18

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)
```



SHELL:

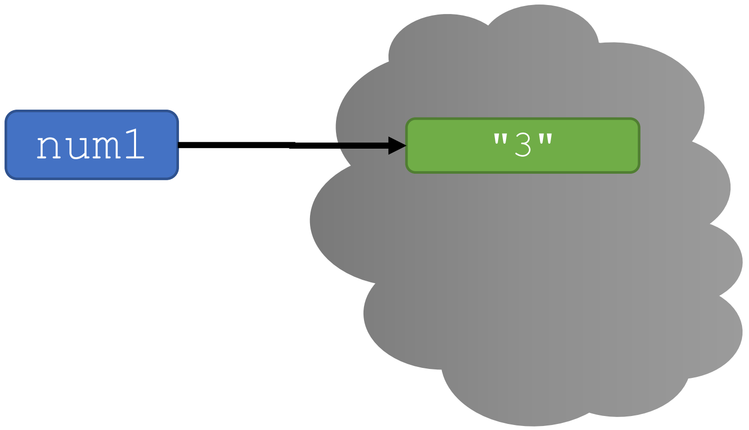
```
Type anything: howdy
howdyhowdyhowdyhowdyhowdy
```

INPUT

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

```
num1 = input("Type a number: ")
print(5*num1)
num2 = int(input("Type a number: "))
print(5*num2)
```

"3"



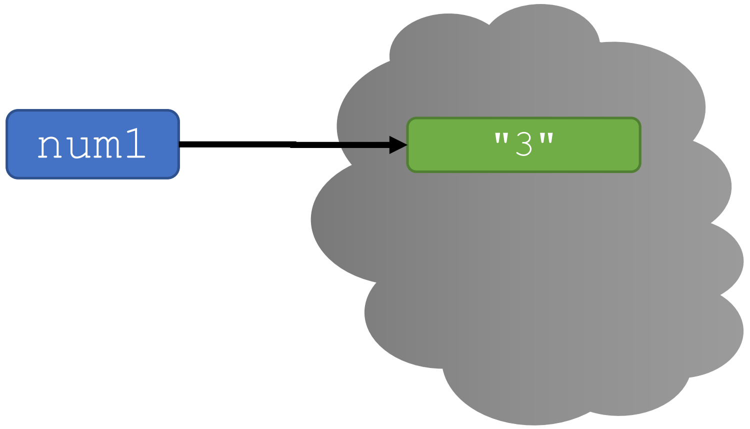
SHELL:

```
Type a number: 3
```

INPUT

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

```
num1 = input("Type a number: ")
print(5*num1)
num2 = int(input("Type a number: "))
print(5*num2)
```



SHELL:

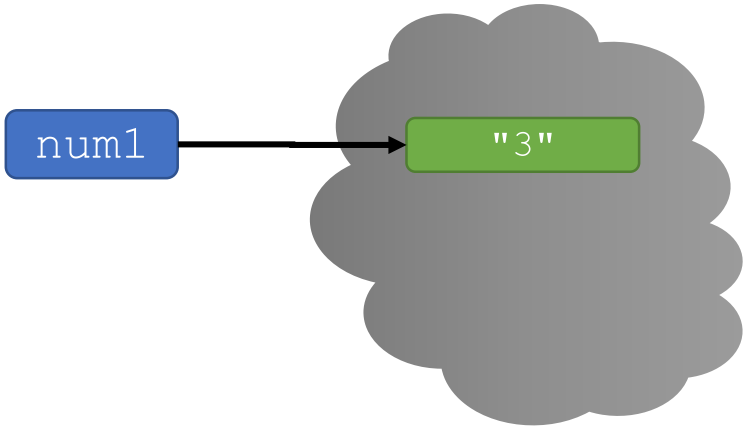
```
Type a number: 3
33333
```

INPUT

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

```
num1 = input("Type a number: ")
print(5*num1)
num2 = int(input("Type a number: "))
print(5*num2)
```

"3"



SHELL:

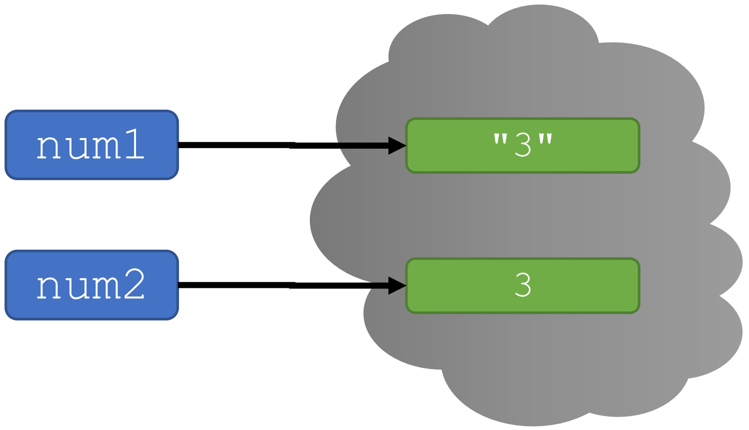
```
Type a number: 3
33333
Type a number: 3
```

INPUT

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

```
num1 = input("Type a number: ")
print(5*num1)
num2 = int(input("Type a number: "))
print(5*num2)
```

3



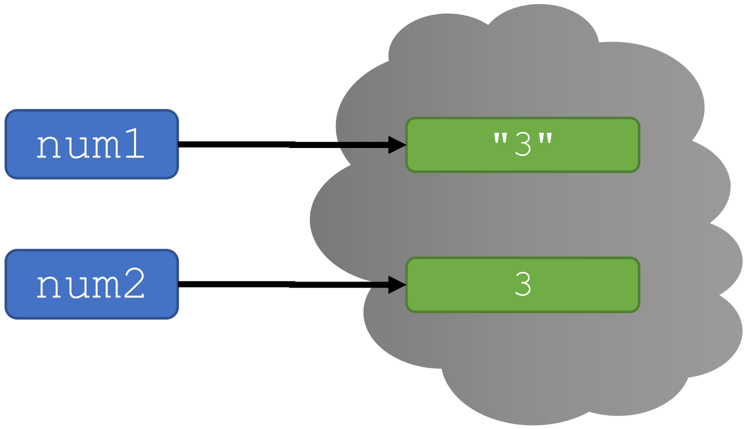
SHELL:

```
Type a number: 3
33333
Type a number: 3
```

INPUT

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

```
num1 = input("Type a number: ")
print(5*num1)
num2 = int(input("Type a number: "))
print(5*num2)
```



SHELL:

```
Type a number: 3
33333
Type a number: 3
15
```

YOU TRY IT!

- Write a program that
  - Asks the user for a verb
  - Prints “I can _ better than you” where you replace _ with the verb.
  - Then prints the verb 5 times in a row separated by spaces.
  - For example, if the user enters `run`, you print:

```
I can run better than you!
run run run run run
```

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
  - next_guess = guess - $\frac{f(guess)}{f'(guess)}$
- Partial code of algorithm that gets input and finds next guess

```
#Try Newton Raphson for cube root
x = int(input('What x to find the cube root of? '))
g = int(input('What guess to start with? '))
print('Current estimate cubed = ', g**3)
                    f(g)               f'(g)
next_g = g - ((g**3 - x)/(3*g**2))
print('Next guess to try = ', next_g)
```

F-STRINGS

- Available starting with Python 3.6
- Character `f` followed by a **formatted string literal**
  - Anything that can be appear in a normal string literal
  - Expressions bracketed by curly braces { }
- Expressions in curly braces evaluated at runtime, automatically converted to strings, and concatenated to the string preceding them

Introduces an extra space

```
num = 3000
fraction = 1/3
print(num*fraction, 'is', fraction*100, '% of', num)
print(num*fraction, 'is', str(fraction*100) + '% of', num)
print(f'{num*fraction} is {fraction*100}% of {num}')
```

expressions

27

BIG IDEA

Expressions can be placed anywhere.

Python evaluates them!

CONDITIONS for BRANCHING

BINDING VARIABLES and VALUES

- In CS, there are two **notions of equal**
  - Assignment and Equality test
- `variable = value`
  - **Change the stored value** of variable to value
  - Nothing for us to solve, computer just does the action
- `some_expression == other_expression`
  - A **test for equality**
  - No binding is happening
  - Expressions are replaced by values and computer just does the comparison
  - Replaces the **entire line** with `True` or `False`

COMPARISON OPERATORS

- `i` and `j` are variable names
  - They can be of type ints, float, strings, etc.
- Comparisons below evaluate to the type **Boolean**
  - The Boolean type only has 2 values: `True` and `False`

`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'

31

LOGICAL OPERATORS on bool

- `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

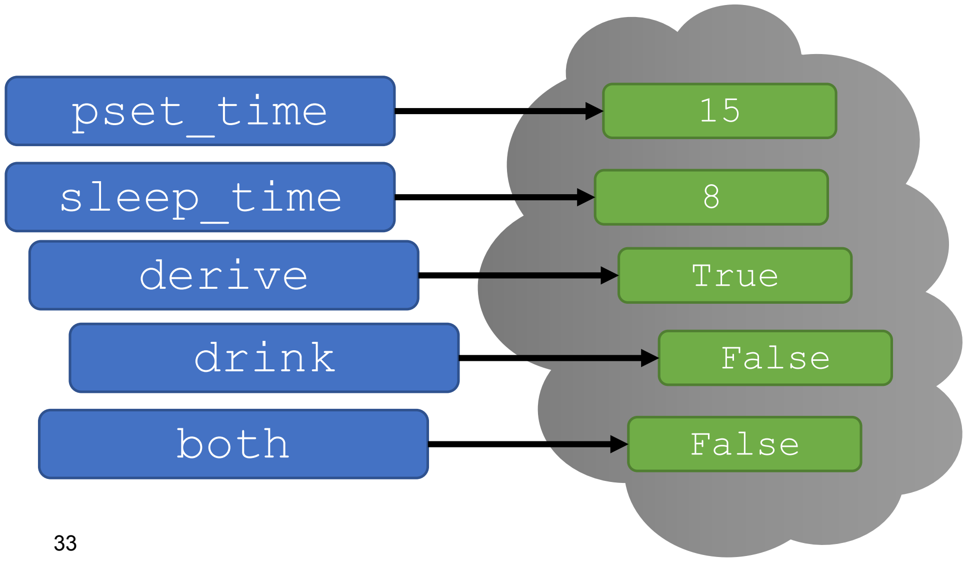
32

COMPARISON EXAMPLE

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

Prints the boolean False

Prints the boolean False



YOU TRY IT!

- Write a program that
  - Saves a secret number in a variable.
  - Asks the user for a number guess.
  - Prints a bool `False` or `True` depending on whether the guess matches the secret.

WHY bool?

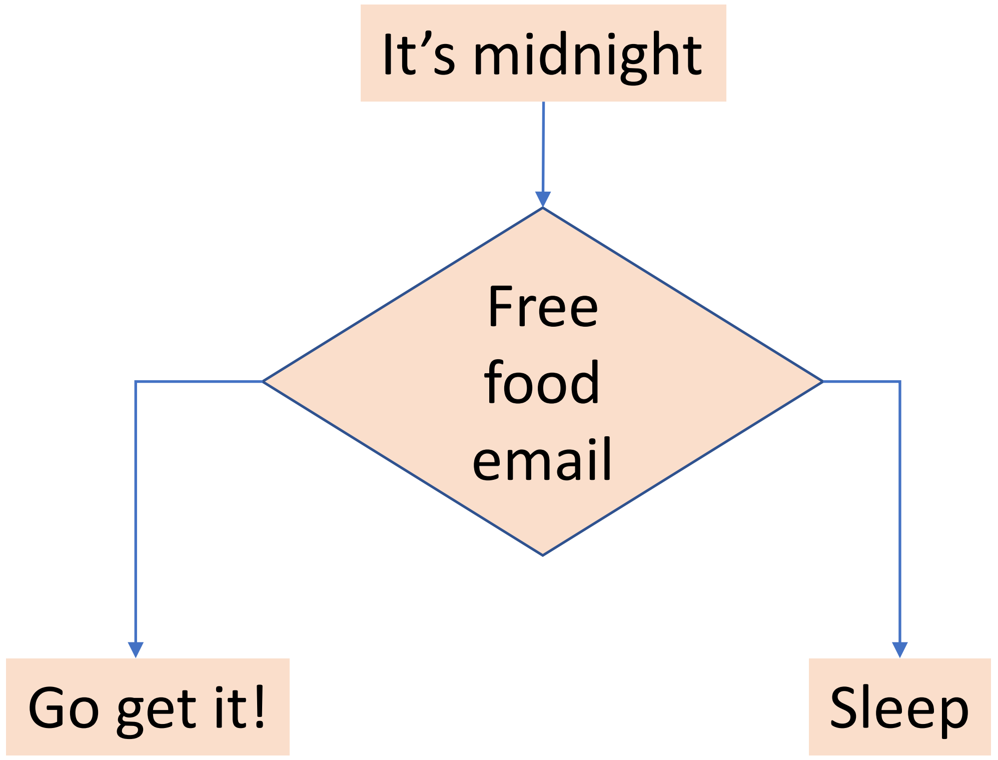
- When we get to flow of control, i.e. branching to different expressions based on values, we need a way of knowing if a condition is true
- E.g., if $\boxed{\text{something is true}}$, $\boxed{\text{do this}}$, otherwise $\boxed{\text{do that}}$

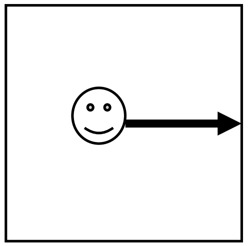
Boolean

Some commands

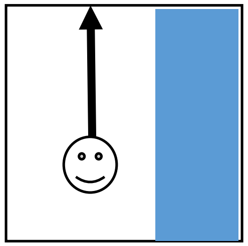
Some other commands

INTERESTING ALGORITHMS INVOLVE DECISIONS

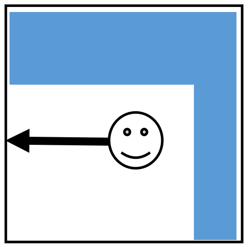




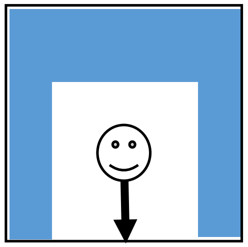
If right clear, go right



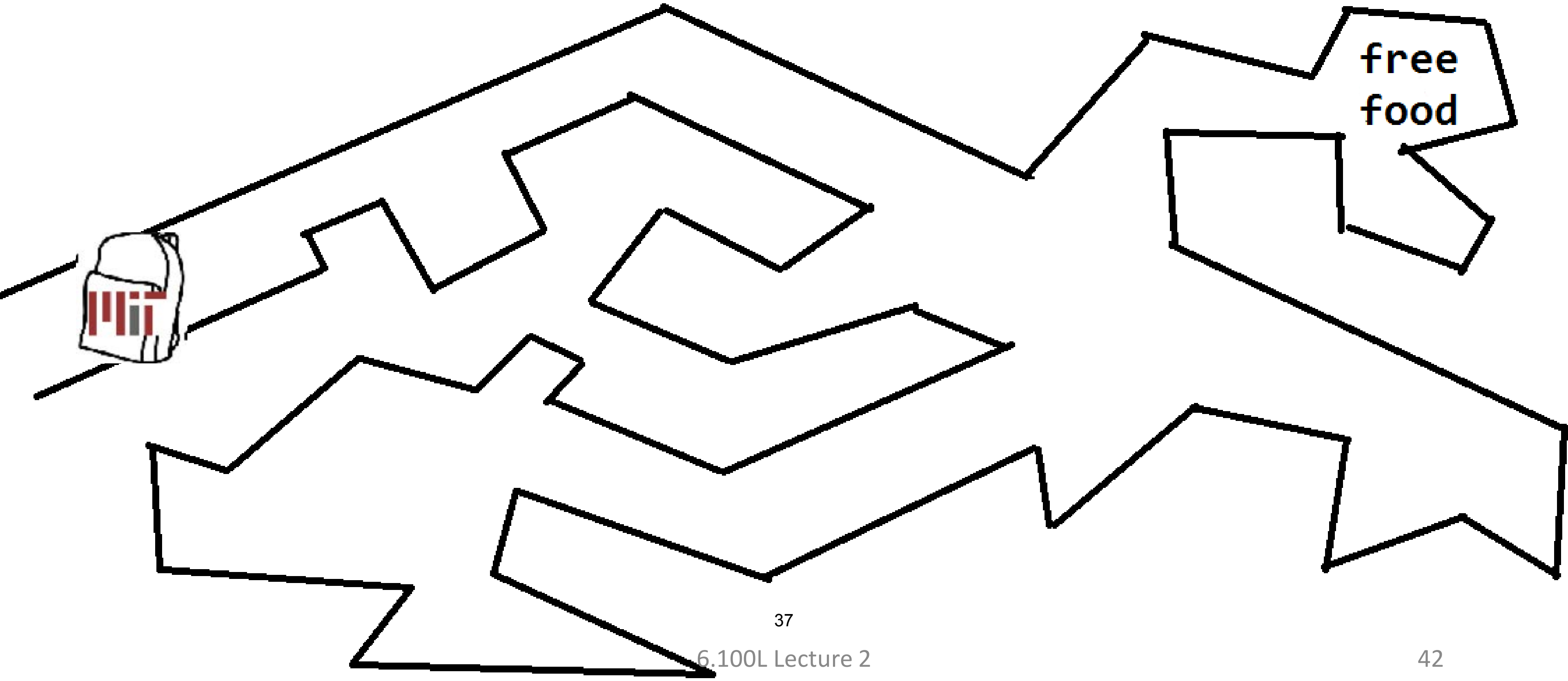
If right blocked, go forward



If right and front blocked, go left



If right , front, left blocked, go back



BRANCHING IN PYTHON

```
if <condition>:
    <code>
    <code>
    ...
<rest of program>
```

- `<condition>` has a value `True` or `False`
- **Indentation matters** in Python!
- Do code within if block if condition is `True`

BRANCHING IN PYTHON

```
if <condition>:
    <code>
    <code>
    ...
<rest of program>
```

```
if <condition>:
    <code>
    <code>
    ...
else:
    <code>
    <code>
    ...
<rest of program>
```

- `<condition>` has a value `True` or `False`
- **Indentation matters** in Python!
- Do code within if block when condition is `True` **or** code within else block when condition is `False`

BRANCHING IN PYTHON

```
if <condition>:
    <code>
    <code>
    ...
<rest of program>
```

```
if <condition>:
    <code>
    <code>
    ...
else:
    <code>
    <code>
    ...
<rest of program>
```

```
if <condition>:
    <code>
    <code>
    ...
elif <condition>:
    <code>
    <code>
    ...
elif <condition>:
    <code>
    <code>
    ...
<rest of program>
```

- `<condition>` has a value `True` or `False`
- **Indentation matters** in Python!
- Run the **first block** whose corresponding `<condition>` is `True`

BRANCHING IN PYTHON

```
if <condition>:
    <code>
    <code>
    ...
<rest of program>
```

```
if <condition>:
    <code>
    <code>
    ...
else:
    <code>
    <code>
    ...
<rest of program>
```

```
if <condition>:
    <code>
    <code>
    ...
elif <condition>:
    <code>
    <code>
    ...
elif <condition>:
    <code>
    <code>
    ...
<rest of program>
```

```
if <condition>:
    <code>
    <code>
    ...
elif <condition>:
    <code>
    <code>
    ...
else:
    <code>
    <code>
    ...
<rest of program>
```

- `<condition>` has a value `True` or `False`
- **Indentation matters** in Python!
- Run the **first block** whose corresponding `<condition>` is `True`. The else block runs when no conditions were `True`

BRANCHING EXAMPLE

```
pset_time = ???
sleep_time = ???
if (pset_time + sleep_time) > 24:
    print("impossible!")
elif (pset_time + sleep_time) >= 24:
    print("full schedule!")
else:
    leftover = abs(24-pset_time-sleep_time)
    print(leftover,"h of free time!")
print("end of day")
```

Condition that evaluates to a Boolean

This indented code executed if line above is True

This indented code executed if line above is True and the if condition is False

This else block runs only if previous conditions were all False

YOU TRY IT!

- Semantic structure matches visual structure
- Fix this buggy code (hint, it has bad indentation)!

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

INDENTATION and NESTED BRANCHING

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

```
x = float(input("Enter a number for x: "))   5     5     0
y = float(input("Enter a number for y: "))   5     0     0
if x == y:                                   True  False True
    print("x and y are equal")               <-          <-
    if y != 0:                               True        False
        print("therefore, x / y is", x/y)    <-
elif x < y:                                        False
    print("x is smaller")
else:
    print("y is smaller")                          <-
print("thanks!")                             <-    <-    <-
```

BIG IDEA

Practice will help you build a mental model of how to trace the code

Indentation does a lot of the work for you!

YOU TRY IT!

- What does this code print with
  - y = 2
  - y = 20
  - y = 11
- What if `if x <= y:` becomes `elif x <= y:` ?

```
answer = ''
x = 11
if x == y:
    answer = answer + 'M'
if x >= y:
    answer = answer + 'i'
else:
    answer = answer + 'T'
print(answer)
```

YOU TRY IT!

- Write a program that
  - Saves a secret number.
  - Asks the user for a number guess.
  - Prints whether the guess is too low, too high, or the same as the secret.

BIG IDEA

Debug early, debug often.

Write a little and test a little.

Don’t write a complete program at once. It introduces too many errors.

Use the Python Tutor to step through code when you see something unexpected!

SUMMARY

- Strings provide a new data type
  - They are **sequences of characters**, the **first one at index 0**
  - They can be indexed and sliced
- Input
  - Done with the **`input`** command
  - Anything the user inputs is **read as a string object**!
- Output
  - Is done with the **`print`** command
  - Only objects that are printed in a .py code file will be **visible in the shell**
- Branching
  - Programs execute **code blocks** when conditions are true
  - In an `if-elif-elif`… structure, the **first condition that is True** will be executed
  - **Indentation matters** in Python!