Introduction to

PYTHON

Demystifying the World of Artificial Intelligence and Exploring Its Potential

ASPECTS OF LANGUAGES

- **semantics** is the meaning associated with a syntactically correct string of symbols with no static semantic errors
  - English: can have many meanings "*Flying planes can be dangerous*"
  - programming languages: 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 semantic errors but **different meaning than what programmer intended**
  - program crashes, stops running
  - program runs forever
  - program gives an answer but different than expected

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

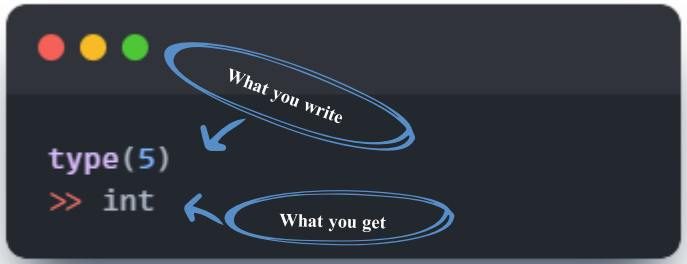
OBJECTS

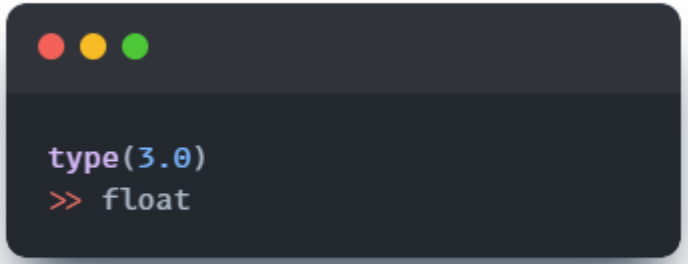
- Programs manipulate **data objects**

- Objects have a **type** that defines the kinds of things
- programs can do to them
  - Ana is a human so she can walk, speak English, etc.
  - Chewbacca is a wookie so he can walk, “mwaaarhrhh”, etc.
- Objects are
  - scalar (cannot be subdivided)
  - non-scalar (have internal structure that can be accessed)

SCALAR OBJECTS

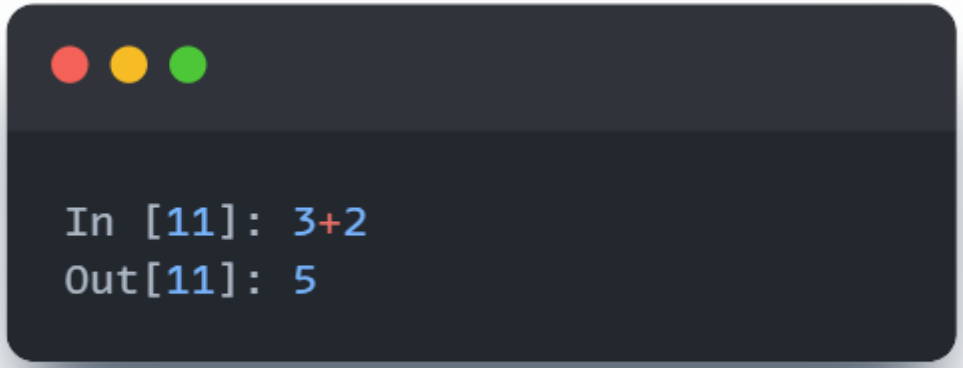
- int – represent **integers**, ex. 5
- float – represent **real numbers**, ex. 3.27
- bool – represent **Boolean** values True and False
- NoneType – **special** and has one value, None
- can use type() to see the type of an object



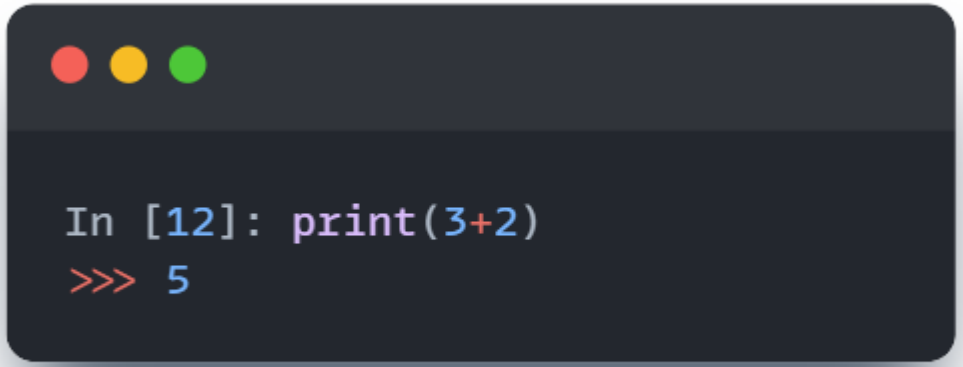


PRINTING TO CONSOLE

- to show output from code to a user, use **print** command



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



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

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 float

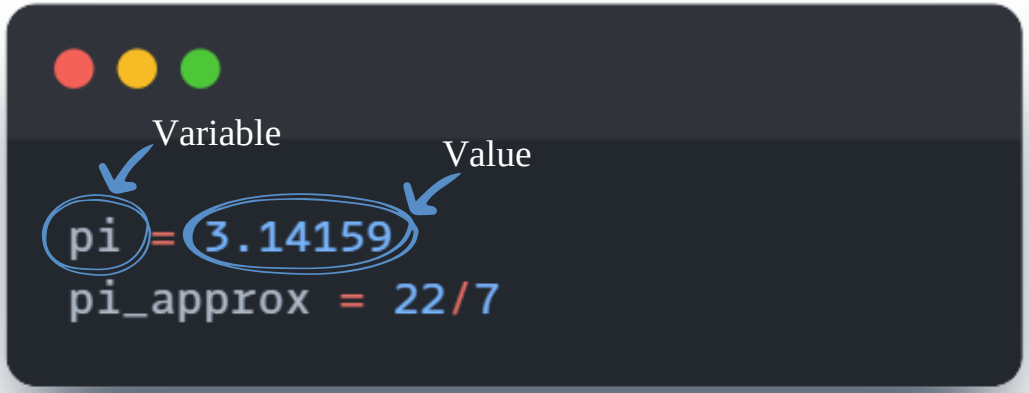
- **i%j** → the **remainder** when i is *divided by* j
- **i**j** → i to the **power** of j

SIMPLE OPERATIONS

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

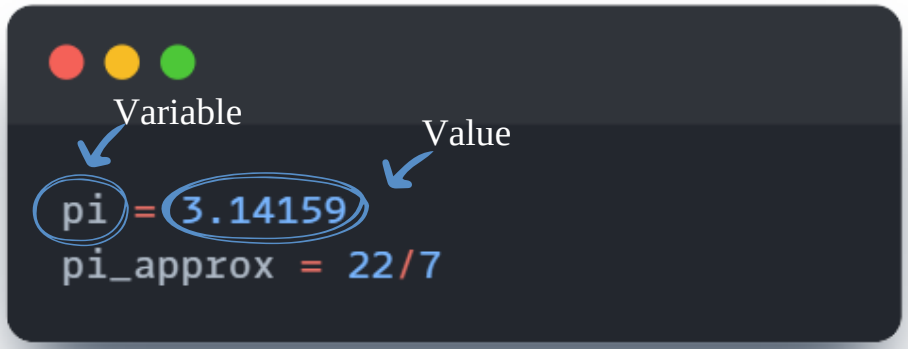
BINDING VARIABLES AND VALUES

- **equal sign is an assignment of a value to a variable name**



BINDING VARIABLES AND VALUES

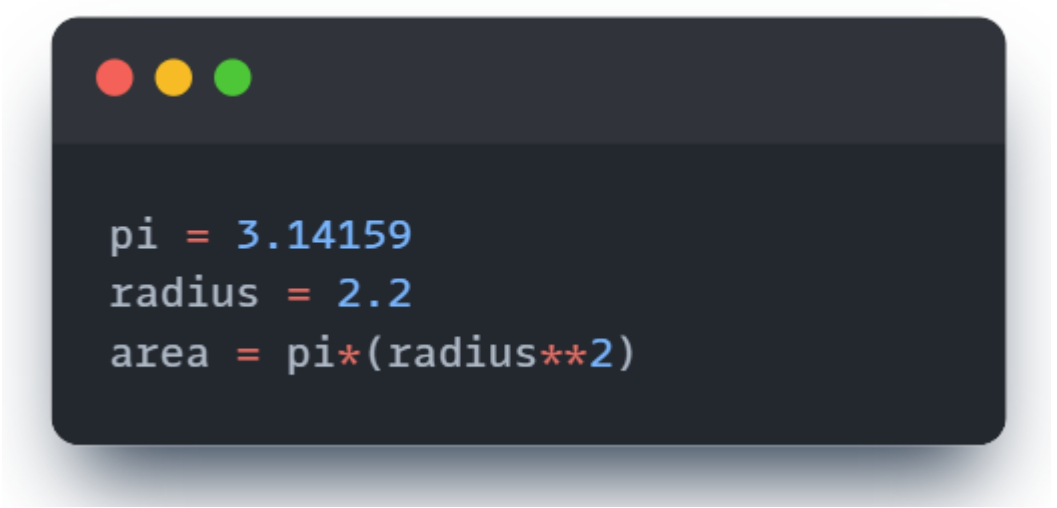
- equal sign is an assignment of a value to a variable name



- value stored in computer memory
- an assignment binds name to value
- retrieve value associated with name or variable by invoking the name, by typing pi

ABSTRACTING EXPRESSIONS

- why give names to values of expressions?
- to reuse names instead of values
- easier to change code later



```
pi = 3.14159
radius = 2.2
area = pi*(radius**2)
```

PROGRAMMING vs MATH

- **in programming, you do not “solve for x”**

```
pi = 3.14159
radius = 2.2

# area of circle
area = pi*(radius**2)
radius = radius+1
```

an assignment (=)
* expression on the right, evaluated to a value
* variable name on the left
* equivalent expression to radius radius + 1
is radius += 1

CHANGING BINDINGS

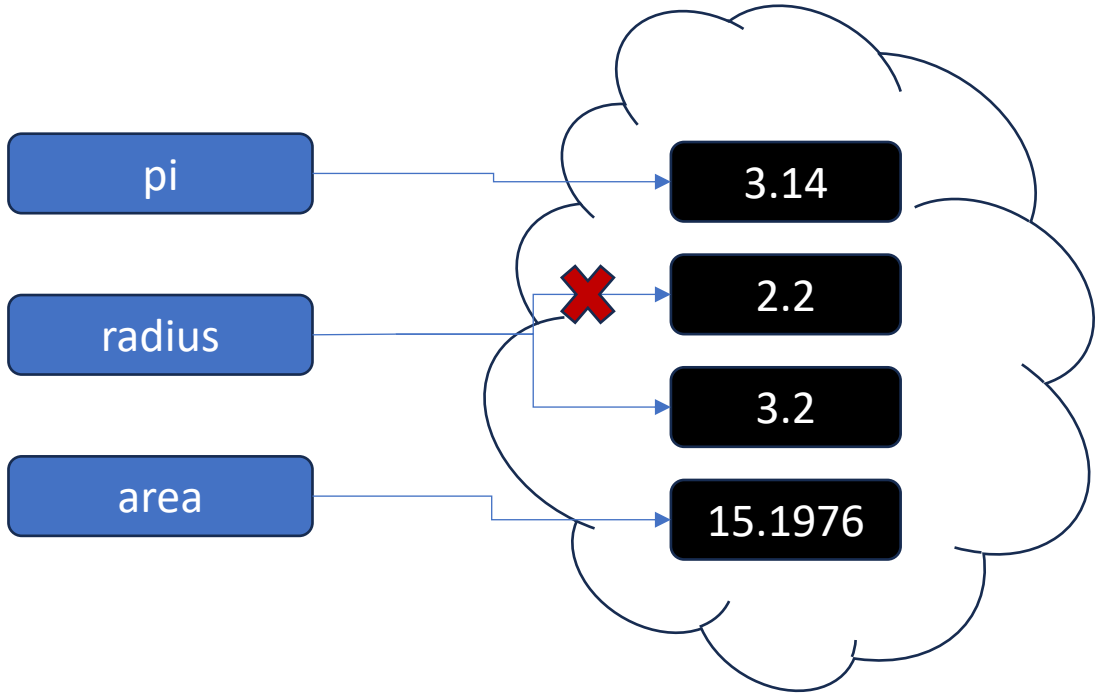
- can **re-bind** variable names using new assignment statements
- previous value may still stored in memory but lost the handle for it
- value for area does not change until you tell the computer to do the calculation again

CHANGING BINDINGS

- 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 = 3.14159
radius = 2.2

# area of circle
area = pi*(radius**2)
radius = radius+1
```



STRINGS

- letters, special characters, spaces, digits
- enclose in **quotation marks or single quotes**

  hi = "hello there"
- **concatenate** strings

  name = "ana"

  greet = hi + name

  greeting = hi + " " + name
- do some **operations** on a string as defined in Python docs

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

INPUT/OUTPUT: print

- used to **output** stuff to console
- keyword is print

```
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)
```

INPUT/OUTPUT: print

- prints whatever is in the quotes
- user types in something and hits enter
- binds that value to a variable

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

- input gives you a string so must cast if working with numbers

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

COMPARISON OPERATORS ON (*int, float, string*)

- 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

LOGIC 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)
derive = True
drink = False
both = drink and derive
print(both)
```

CONTROL FLOW - BRANCHING

```
if <condition>:
    <expression>
    <expression>
    ...
```

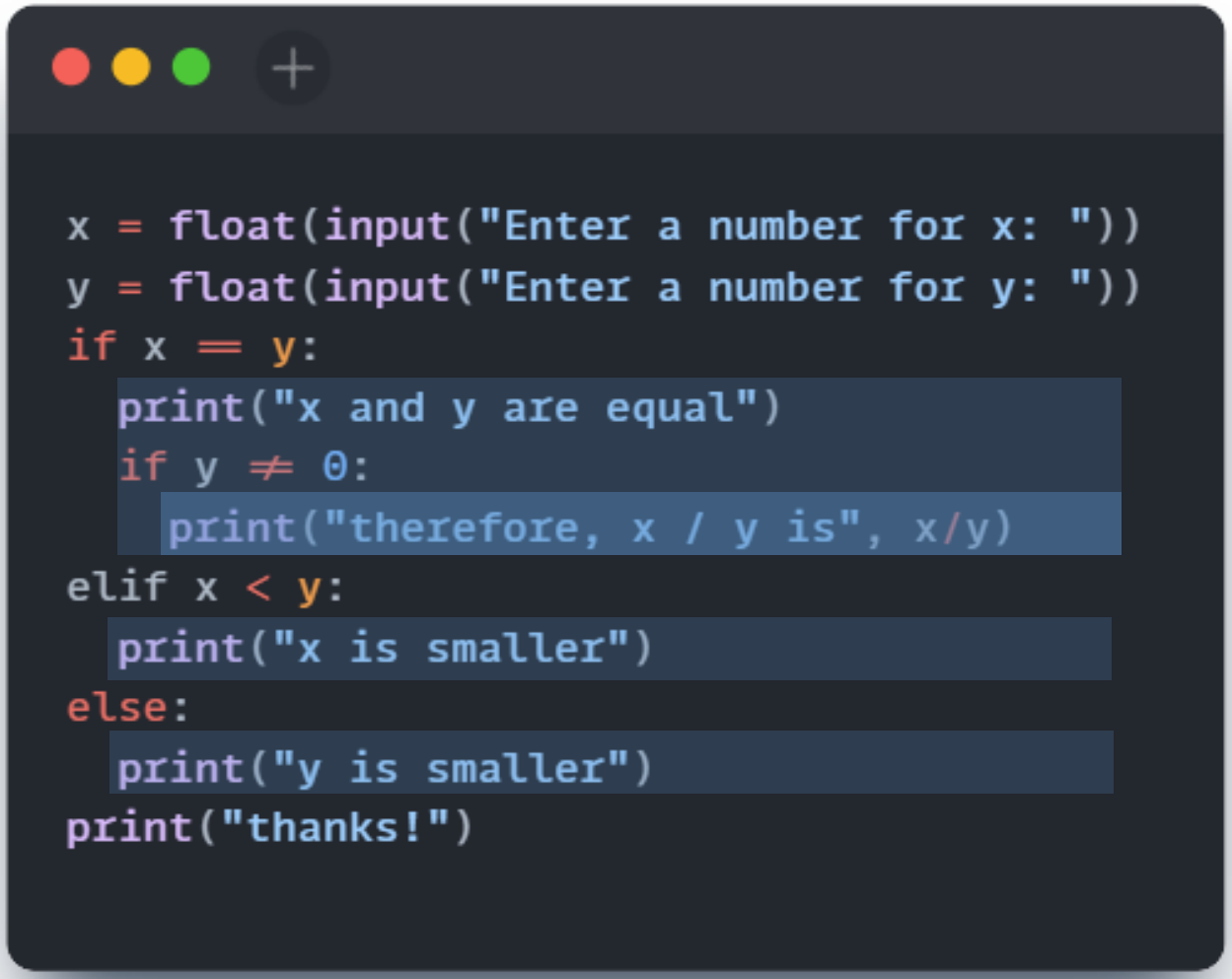
```
if <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```

```
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

- **matters in Python**
- **how you denote blocks of code**



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

CONTROL FLOW: while LOOPS

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

- <*condition*> **evaluates to a Boolean**
- if <*condition*> **is** True, **do all the steps inside the**
- **while code block**
- **check** <*condition*> **again**
- **repeat until** <*condition*> **is** False

while LOOP EXAMPLE

```
You are in the Lost Forest.
************
************
   ☺
************
************
Go left or right?
```

- **PROGRAM:**

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

CONTROL FLOW: while and for LOOPS

- iterate through numbers in a sequence

```
# more complicated with while loop
n = 0
while n < 5:
print(n)
n = n+1

# shortcut with for loop
for n in range(5):
print(n)
```

CONTROL FLOW: while 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**.

Range (*start, stop, step*)

- **default values are** start = 0 **and** step = 1 **and optional**
- **loop until value is** 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)
```

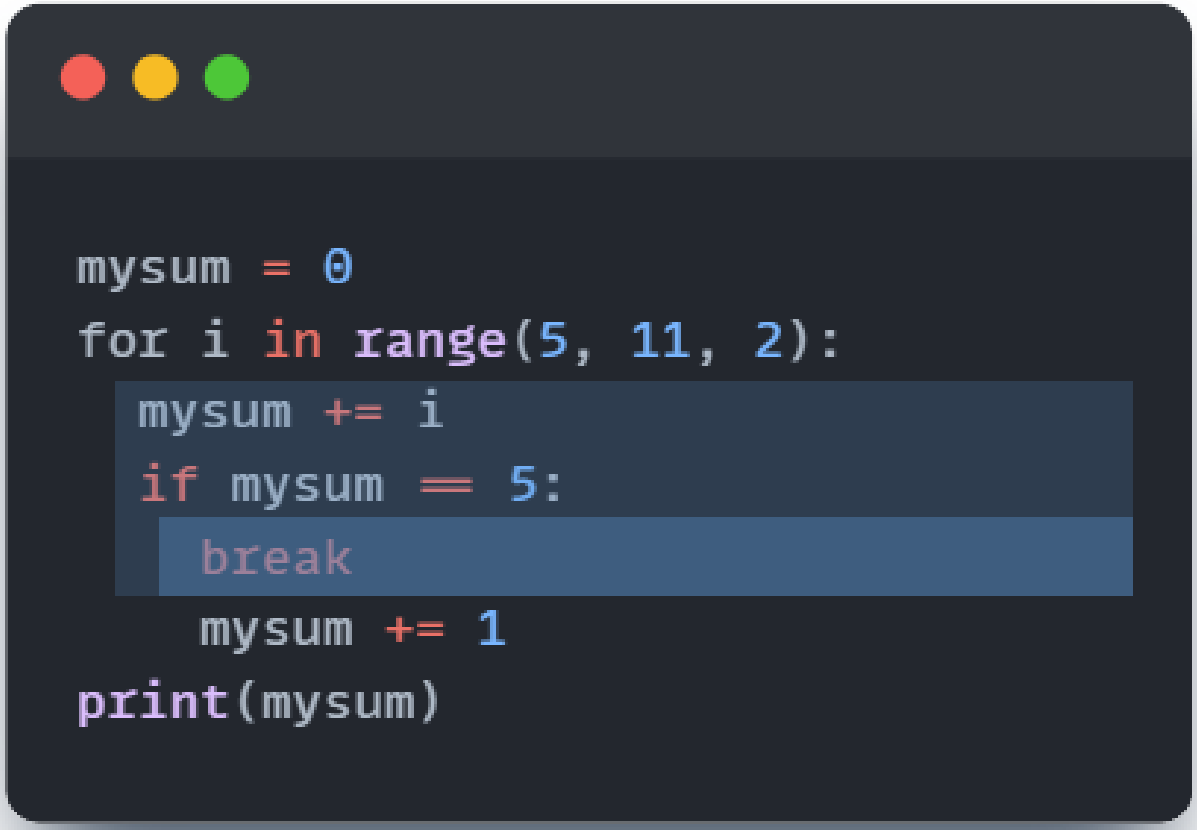
break STATEMENT

- **immediately exits** whatever loop it is in
- **skips** remaining expressions in code block
- **exits** only innermost loop!

```
while <condition_1>:
    while <condition_2>:
        <expression_a>
        break
        <expression_b>
    <expression_c>
```

break STATEMENT

- what happens in this program?



```
mysum = 0
for i in range(5, 11, 2):
    mysum += i
    if mysum == 5:
        break
    mysum += 1
print(mysum)
```

COMPARISON OPERATORS ON (*int, float, string*)

for

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

while

- 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

- think of as a **sequence** of case sensitive characters
- can compare strings with ==, >, < etc.
- 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 → 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"
```

STRINGS

- can slice strings using [*start:stop:step*]
- if give two numbers, [*start:stop*], *step=1* by default
- you can also omit numbers and leave just colons

```
s = "abcdefgh"
```

`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 `"hgfedbca"`, same as `s[-1:-(len(s)+1):-1]`

`s[4:1:-2]` → evaluates to `"ec"`

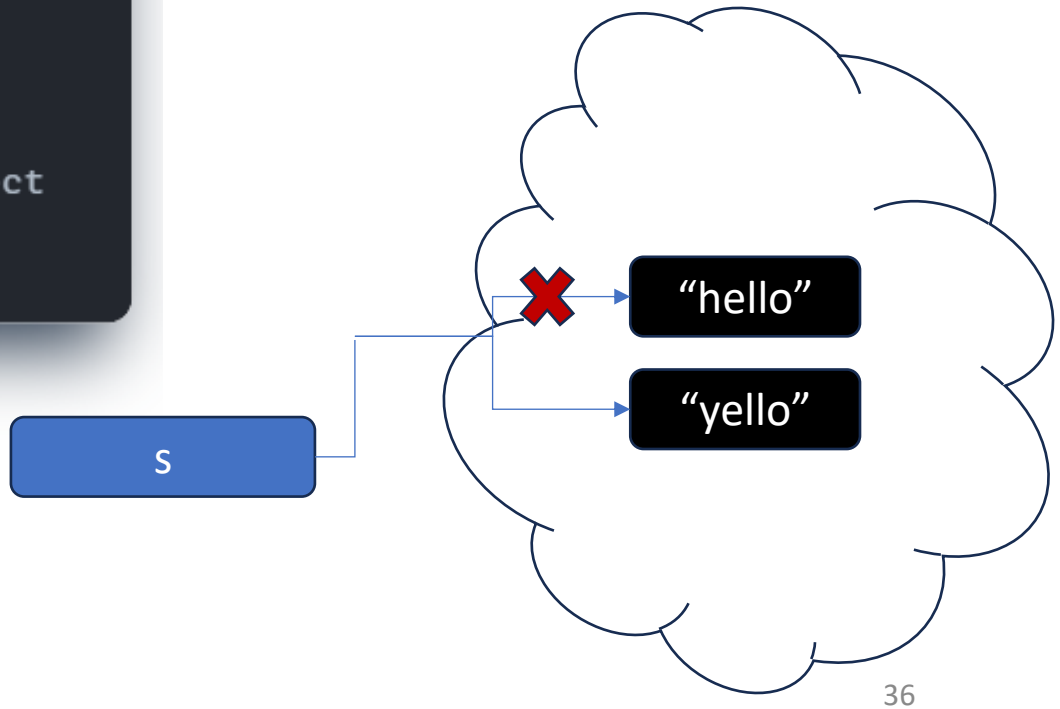
STRINGS

- strings are "**immutable**" – cannot be modified

```
s = "hello"

s[0] = 'y'   ⟶ gives an error

s = 'y'+s[1:len(s)] ⟶  is allowed,s bound to new object
```



Advanced string methods

- **string.split(separator, maxsplit):** splits a string and returns a list of elements separated by a delimiter.
  - **Separator** is the delimiter such as space, comma,etc.
  - **Maxsplit** is the maximum number of splitted elements and the rest will be combined in one element

```
saluteStr = "hellojohnhijohnbonjourjohnbuongiorno"
print(saluteStr.split('john'))

>> ['hello', 'hi', 'bonjour', 'buongiorno']
```

Advanced string methods

- **string. find(searchText,[starting_position, [ending_position]]):**

search the string for some text between two positions.

  - **searchText**: the text you are searching for.
  - **starting_position**: the position inside the string where you want the search to start at
  - **ending_position**: the position inside the string where you want the search to end at

Advanced string methods

- **string.replace(search_string,replace_string [,counter]):**

replaces a substring with another substring inside the bigger string.

  - **search_string**: the substring to search for to be replaced.
  - **replace_string**: substring to use as a replacement
  - **counter**: the number of matches you want to replace substrings found

Advanced string methods

- **Upper**(): convert to all uppercase
- **Lower**(): convert to all lowercase
- **Capitalize**(): first letter capital and rest lowercase

Advanced string methods

- **string.count(search_text [, start [,end]]):**

counts the number of occurnces of substring inside bigger string.

  - **searchText**: the text you are searching for.
  - **starting_position**: the position inside the string where you want the search to start at
  - **ending_position**: the position inside the string where you want the search to end at

Advanced string methods

- **len**(): returns length of string in characters

Advanced string methods

- **Logical methods starting with is**


```
>>> 11 ".isspace()
True

>>> "1234abcd".isalnum()
True
>>> "1234abcd".isalpha()
False

>>> "123445".isnumeric()
True

>>> "this text".isupper()
False
>>> "this text".islower()
True
```

for LOOPS RECAP

- for loops have a **loop variable** that iterates over a set of values

```
for var in range(4):
    <expressions>
```

→ `var` iterates over values 0,1,2,3

→ expressions inside loop executed with each value for `var`

```
for var in range(4,6):
    <expressions>
```

→ `var` iterates over values 4,5

- range is a way to iterate over numbers, but a for loop variable can **iterate over any set of values**, not just numbers!

STRINGS AND LOOPS

- these two code snippets do the **same** thing

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

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

More "**pythonic**"

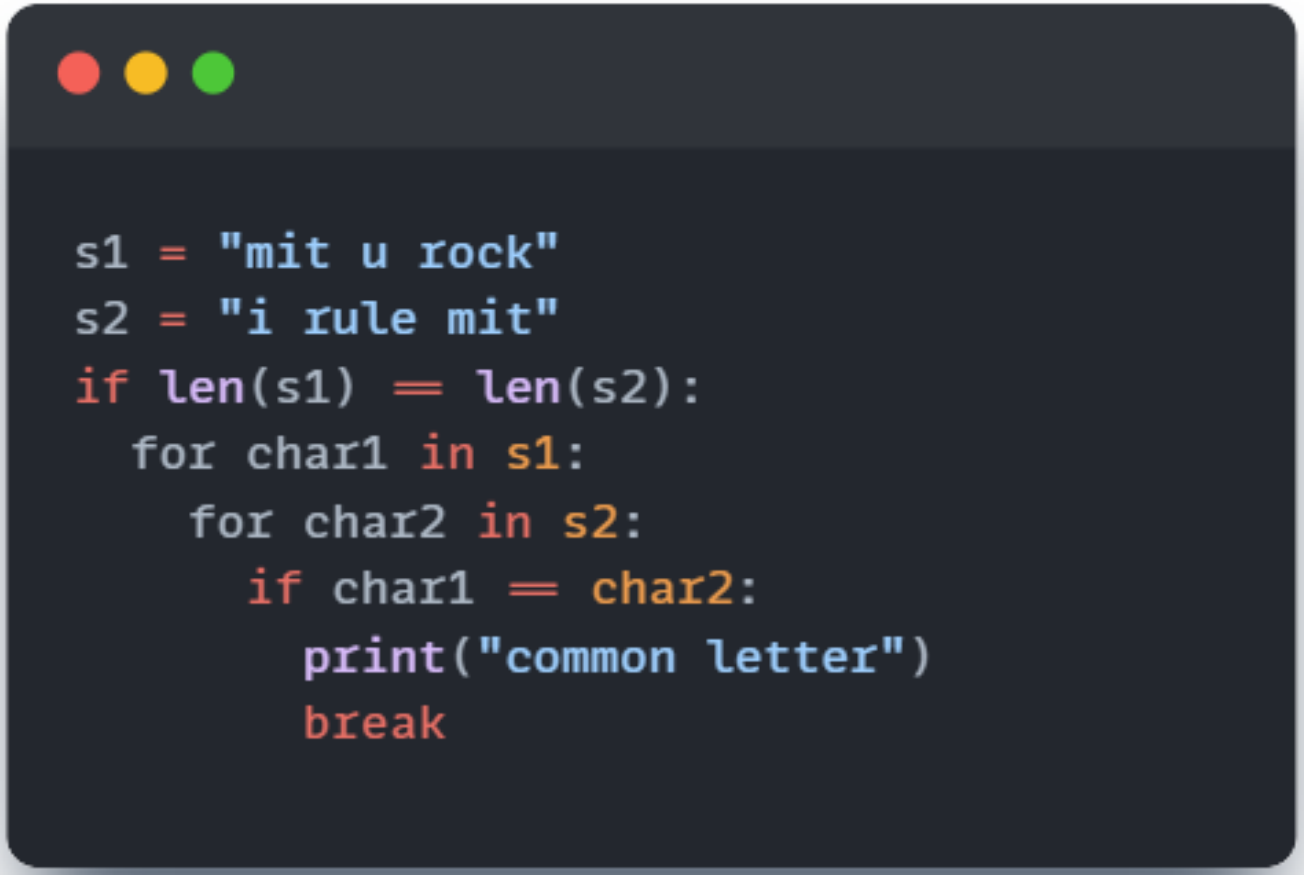
CODE EXAMPLE:

ROBOT CHEERLEADERS

```
an_letters = "aefhilmnorsxAEFHILMNORSX"
word = input("I will cheer for you! Enter a word: ")
times = int(input("Enthusiasm level (1-10): "))
i = 0
while i < len(word):
  char = word[i]
  if char in an_letters:
      print("Give me an " + char + "! " + char)
  else:
      print("Give me a " + char + "! " + char)
  i += 1
print("What does that spell?")
for i in range(times):
print(word, "!!!")
```

for char in word:

EXERCISE



```
s1 = "mit u rock"
s2 = "i rule mit"
if len(s1) == len(s2):
  for char1 in s1:
    for char2 in s2:
      if char1 == char2:
        print("common letter")
        break
```

THANK YOU!

Any Question!