MORE ON PYTHON

LECTURE 01–2 PART 1
THOUGHTS ON LAB
MORE ON PYTHON SCRIPTING

JIM FIX, REED COLLEGE CSCI 121

ADDITIONAL COURSE STAFF

- Tutors:
  - ✦ Yulin Meng
  - ✦ Charlotte Brohme-Chen
  - ✦ Grey Nielson
  - ✦ Kellyn Cowger
  - ➡ Will staff **Library 340** *7-9pm Su through Th*. **I'm still setting this up**.
  - ➡ Some might also attend lab.

- Teaching consultant:
  - ✦ Fatima Campbell
  - ➡ Will observe lectures and some labs.
  - ➡ Setting up anonymous feedback form.
  - ➡ Will take survey around/after week 6.

READINGS

- This week's lecture material can be supplemented with:
  - **Reading:** PP 1.1-1.3; TP Ch. 1 and 2; CP Ch 1.1-1.2
- At the end of lecture we'll also look at:
  - ➡ the conditional statement (i.e. `if`)
  - ➡ the boolean values `True` and `False`

READINGS

- This week's lecture material can be supplemented with:
  - **Reading:** PP 1.1-1.3; TP Ch. 1 and 2; CP Ch 1.1-1.2
- At the end of lecture we'll also look at:
  - ➡ the conditional statement (i.e. `if`)
  - ➡ the boolean values `True` and `False`

"Think Python" text

"Composing Programs" text

Prof. Groce's "Principled Programming" text

NOTES FROM LAB

- I thought the first lab went particularly well for everyone.
  - ➡ Next week I'll pair you.
- Some thoughts from lab:
  - navigating in Terminal and your file system
  - using your editor
  - string versus integer versus floating point; Python input
  - Python output
  - function notation
  - working with Gradescope and the autograder

NAVIGATING W/IN TERMINAL, EDITOR, BROWSER, FILESYSTEM

- Advice: name folders without spaces or punctuation
- Make sure Python program file (i.e. ***scripts***) names end with **`.py`**
- Learn Terminal commands **`cd`**, **`cd ..`**, **`cd`** *folder-name*, **`ls`**, **`pwd`**
- You might sometimes be in the Python interpreter instead of Terminal:
  - prompt is **`>>>`** when in Python
  - can leave Python with **`quit()`** or by typing *control*-**`d`**
  - Note: *control*-**`c`** is a way to stop programs running
  - **`python3`** *my-script* **`.py`** command versus **`python3`** command
- Be mindful of where files live.
  - You might e.g. have copies of the same-named script.
- Learn paths for your folders:
  - **`Documents`**, **`Desktop`**, **`Downloads`**, user's *home* (i.e. **`~`**)
- Don't forget to ***Save*** within the editor.

GRADESCOPE AND THE AUTOGRADER

- You have to get prompt text and program output exactly right.
  - ➡ Spelling, spacing, punctuation, line breaks etc. must match exactly.

- If everything looks good…
  - ➡ You'll get a score of 10.0 from the *autograder*. Otherwise 0.0.
  - ➡ We give up to an additional 10.0 points for programming "*style*" and approach.
    - *More on this later…*

- There are visible versus hidden tests:
  - ➡ For visible tests we will reveal what we input to test your code.
  - ➡ For hidden tests we won't reveal these; we'll just tell you "something's not right."
- **Goal:** to get you used to checking the work yourself.

PYTHON OUTPUT

- Suppose the script text below is saved as **print_fun.py**

```
print("Hello there! Isn't this \"fun\" stuff?")
print()
print("Yes\nI\nagree.")
x = " my" * 3
y = 3.5
print("Oh" + x + ". The value is " + str(y) + ".")
```

- On my Mac, within Terminal, after the **prompt**, I enter the **command**:

```
CO2MX1KLFH04:examples jimfix$ python3 print_fun.py
Hello there! Isn't this "fun" stuff?

Yes
I
agree.
Oh my my my. The value is 3.5.
CO2MX1KLFH04:examples jimfix$
```

- The Python interpreter outputs those **six lines of text**.

PYTHON INPUT; VALUE TYPES; CONVERTING TO STRINGS

- Suppose the script text below is saved as **types.py**

```
a = input("Enter A: ")
b = float(input("Enter B: "))
c = int(input("Enter C: "))
print("A is " + a + " and has type " + str(type(a)) + ".")
print("B is " + b + " and has type " + str(type(b)) + ".")
print("C is " + c + " and has type " + str(type(c)) + ".")
```

- Here is what gets output by Python:

```
C02MX1KLFH04:examples jimfix$ python3 types.py
Enter A: 2
Enter B: 2
Enter C: 2
A is 2 and has type <class 'str'>.
A is 2.0 and has type <class 'float'>.
A is 2 and has type <class 'int'>.
C02MX1KLFH04:examples jimfix$
```

INTERACTING WITH THE PYTHON INTERPRETER

- Python can be used to "live script":

```
CO2MX1KLFH04:examples jimfix$ python3
>>> print(“hello”)
hello
>>> print(6 * 7)
42
>>> result = 6 * 7
>>> print(result)
42
>>>
```

- We can try a Python coding by interacting directly with the interpreter.
- We type in Python statements one at a time.
- Each line gets executed immediately.
- Each statement typically performs an action, has an effect.

THE INTERPRETER AS CALCULATOR

- Python can also evaluate expressions; compute and display their result:

```
CO2MX1KLFH04:examples jimfix$ python3
>>> 6 * 7
42
>>> result = 6 * 7
>>> result
42
>>> “hello” + “ “ + “there”
’hello there’
>>>
```

- Here, Python is acting differently. It calculates the value of the expression, then (quietly) converts that value into some readable text characters, then displays that text.

THE INTERPRETER AS CALCULATOR

- Python can be also be used to evaluate expressions:

```
CO2MX1KLFH04:examples jimfix$ python3
>>> 6 * 7
42
>>> result = 6 * 7
>>> result
42
>>> “hello” + “ “ + “there”
’hello there’
>>>
```

- It repeatly applies three steps as a "***R**ead-**E**val-**P**rint **L**oop*" or ***REPL***:
  - ***READ***: it looks at the expression entered after **>>>**
  - ***EVALUATE***: it performs a calculation to obtain a value
  - ***PRINT***: it displays the result as some text.

THE INTERPRETER AS CALCULATOR

- Python can be used to evaluate expressions:

```
CO2MX1KLFH04:examples jimfix$ python3
>>> 6 * 7
42
>>> result = 6 * 7
>>> result
42
>>> “hello” + “ “ + “there”
’hello there’
>>>
```

- This is the “**READ** - **EVALUATE** - **PRINT LOOP**” (or “**REPL**”).
- Having access to a **REPL** for a programming language is wonderful!
- It’s a big reason we teach programming in Python.

PYTHON PROVIDES ARITHMETIC

```
>>> 3 + 7
10
>>> 4 + 2 * 3
10
>>> (4 + 2) * 3
18
>>> 4 / 16
0.25
>>> 2 ** 4
16
>>> 0.1 + 0.2
0.30000000000000004
```

PYTHON PROVIDES ARITHMETIC

```
>>> 3 + 7
10
>>> 4 + 2 * 3
10
>>> (4 + 2) * 3
18
>>> 4 / 16
0.25
>>> 2 ** 4
16
>>> 0.1 + 0.2
0.30000000000000004
>>> type(4)
<class 'int'>
>>> type(0.25)
<class 'float'>
```

INTEGERS VERSUS FLOATING POINT NUMBERS

- Python has two types of number values: `int` and `float`
- With integers, computation is exact.
- With floating point numbers ("*floats*"), computation is approximate.

```
>>> 10 / 2
5.0
>>> 3 + 4.0
7.0
>>> int(8.7)
8
```

INTEGER VERSUS FLOATING POINT DIVISION

- With the normal division operation, the slash `/`, you get a float.

```
>>> 10.2 / 2.0
5.1
>>> 10 / 2
5.0
>>> 87 / 10
8.7
```

INTEGER VERSUS FLOATING POINT DIVISION

- There is also an integer division operation, the double slash operator `//`.
  - ➡ This gives the integer quotient.
  - ➡ The remainder due to the division is discarded.

```
>>> 10 // 2
5
>>> 87 // 10
8
```

PYTHON HAS // AND % OPERATORS

- The `//` operation ("div") gives the integer quotient due to the division of two integers:

```
>>> 345 // 12
28
```

- The `%` operation ("mod") gives the integer remainder due to the division of two integers:

```
>>> 345 % 12
9
```

- This property always holds:
  - ➡ *number* = *quotient* x *divisor* + *remainder*

```
>>> 28 * 12 + 9
345
```

EXAMPLE USES

```
>>> 345 % 10
?????????
>>> 345 // 10
?????????
>>> 6789 % 2
?????????
>>> 6790 % 2
?????????
>>> -26 % 2
?????????
>>> -76 % 10
?????????
>>> -26 // 2
?????????
>>> -76 // 10
?????????
```

EXAMPLE USES

```
>>> 345 % 10
5
>>> 345 // 10
34
>>> 6789 % 2
1
>>> 6790 % 2
0
>>> -26 % 2
0
>>> -76 % 10
4
>>> -26 // 2
-13
>>> -76 // 10
-8
```

PYTHON CAN APPLY KNOWN FUNCTIONS...

```
>>> pow(2,3)
8
>>> abs(-3)
3
>>> abs(4 + 2)
6
>>> min(3,7)
3
>>> max(4, 10.5 + 8.3, 6)
18.8
>>> from math import sqrt, pow
>>> sqrt(2.0)
1.4142135623730951
>>> pow(2.0,4.5)
22.627416997969522
```

TEXT STRINGS

- Python can store and compute with text:

```
>>> entry = input("Enter something: ")
Enter something: some thing
>>> entry
'some thing'
>>> type(entry)
<class 'str'>
>>> "hello"
'hello'
>>> type("hello")
<class 'str'>
>>> 'hello'
'hello'
>>> len(entry)
10
>>> len("hello")
10
```

- To describe a string of characters, you put those literal characters between double quotes.
- You can also use single quotes, and Python chooses to report strings that way,
  - ➡ These distinguish the text from a variable name.

STRING ARITHMETIC

```
>>> “hello” + “there”
‘hellothere’
>>> x = “hello”
>>> x = x + “ there”
>>> x
‘hello there’
>>> x = x + “ i must”
>>> x = x + “ be going”
>>> x
‘hello there i must be going’
```

STRING ARITHMETIC (CONT'D)

```
>>> "hello" * 3
'hellohellohello'
>>> 4 * "hello"
'hellohellohellohello'
>>> "hello" * 0
''
>>>
```

STRING ARITHMETIC (CONT'D)

```
>>> "hello" * 3
'hellohellohello'
>>> 4 * "hello"
'hellohellohellohello'
>>> "hello" * 0
''
>>> "hello" + 3
Error!
>>> "76" + 3
Error!
>>> "76" + str(3)
'763'
>>> int("76") + 3
79
>>> int("hello") + 3
Error!
```

SPECIAL CHARACTERS

- A backslash character `\` followed by a second character expresses special characters
  - ➡ a tab is `\t`, a new line is `\n`, a quote is `\'`, a backslash is `\\`

```
>>> z = input('What\'s your name?')
What's your name?John
>>> “Hello “ + z
'Hello John'
>>> print(“I\’ve “+str(19)+“ characters.\nSee?”)
I've 19 characters.
See?
>>> len(“I\’ve “+str(19)+“ characters.\nSee?”)
19
>>> print(“\thello\nthere”)
    hello
there
>>> print("/\\/\\/\\/\\"
/\/\/\/\
```

BACK TO PYTHON SCRIPTING

- Consider this Python program:

```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("The radius of that circle is "+str(radius)+" units.")
```

BACK TO PYTHON SCRIPTING

- Consider this Python program:

```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("The radius of that circle is "+str(radius)+" units.")
```

- This has is 3 assignment statements and a print statement.
- The first defines/assigns the variable named `pi`.
- The second gets a floating point value (a "calculator number") as input, assigned to `area`. We compute that using an arithmetic formula.
- The functions `float` and `str` convert values of one type to values of another type.

PYTHON EXECUTION

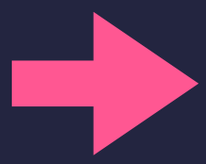
- Let's take a look at this script:

```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

- What we know is that the Python interpreter runs the code, line by line, from the top line to the bottom line.
-

PYTHON EXECUTION

- Let's take a look at this script:

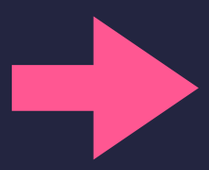


```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

- What we know is that the Python interpreter runs the code, line by line, from the top line to the bottom line.
-

PYTHON EXECUTION

- Let's take a look at this script:



```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

- What we know is that the Python interpreter runs the code, line by line, from the top line to the bottom line.
-

PYTHON EXECUTION

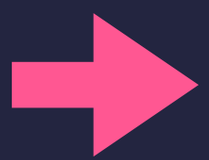
- Let's take a look at this script:

```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

- What we know is that the Python interpreter runs the code, line by line, from the top line to the bottom line.

PYTHON EXECUTION

- Let's take a look at this script:



```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

- What we know is that the Python interpreter runs the code, line by line, from the top line to the bottom line.

PYTHON EXECUTION

- Let's take a look at this script:

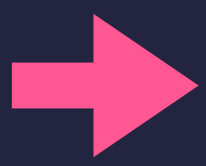
```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

- If you ever want to "watch" a Python program, try out **The Python Tutor**
  https://pythontutor.com/

- *Using it, you'll see something like this...*

PYTHON EXECUTION

- Let's take a look at this script:

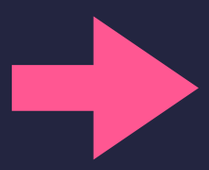


```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

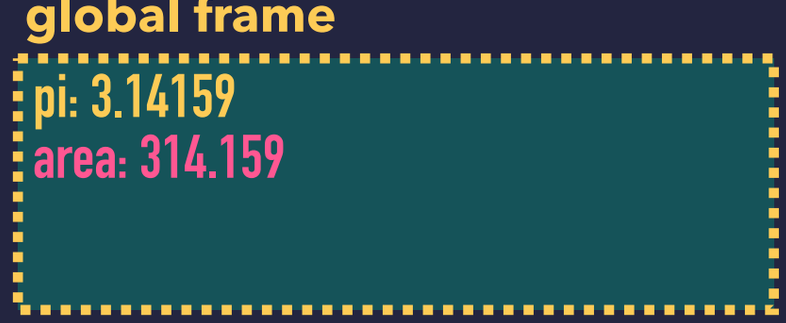
- What we know is that the Python interpreter runs the code, line by line, from the top line to the bottom line.
- It also creates named memory slots for each variable that gets introduced.
  - That named slot stores a calculated value.
  - A variable's associated value is changed with each assignment statement.

PYTHON EXECUTION

- Let's take a look at this script:



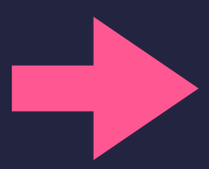
```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```



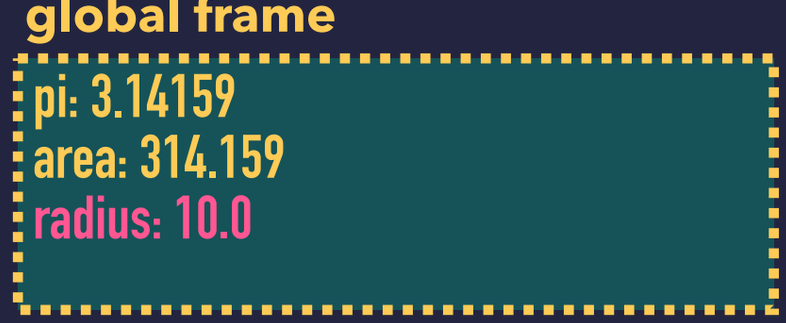
- What we know is that the Python interpreter runs the code, line by line, from the top line to the bottom line.
- It also creates named memory slots for each variable that gets introduced.
  - That named slot stores a calculated value.
  - A variable's associated value is changed with each assignment statement.

PYTHON EXECUTION

- Let's take a look at this script:



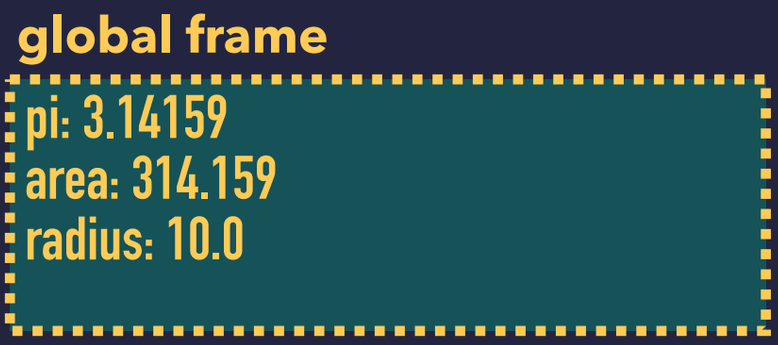
```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```



- What we know is that the Python interpreter runs the code, line by line, from the top line to the bottom line.
- It also creates named memory slots for each variable that gets introduced.
  - That named slot stores a calculated value.
  - A variable's associated value is changed with each assignment statement.

PYTHON EXECUTION

- Let's take a look at this script:

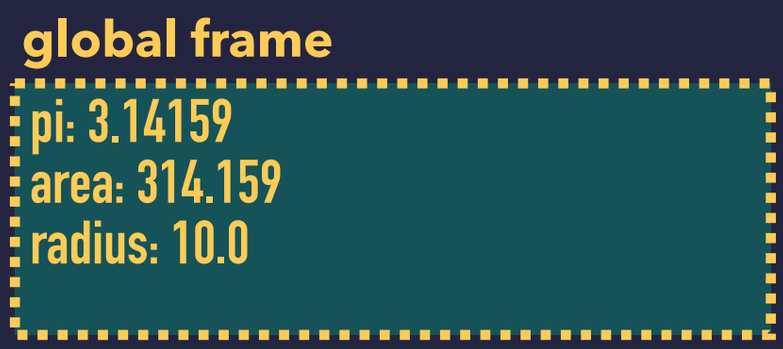



```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

- What we know is that the Python interpreter runs the code, line by line, from the top line to the bottom line.
- It also creates named memory slots for each variable that gets introduced.
  - That named slot stores a calculated value.
  - A variable's associated value is changed with each assignment statement.

PYTHON EXECUTION

- Let's take a look at this script:




```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

- What we know is that the Python interpreter runs the code, line by line, from the top line to the bottom line.
- It also creates named memory slots for each variable that gets introduced.
  - That named slot stores a calculated value.
  - A variable's associated value is changed with each assignment statement.
    - ➡ The collection of variable slots of a script is called the ***global frame***

SUMMARY OF PYTHON SO FAR

- So far, three kinds of statements:
  - `print` statement
  - assignment statement
  - `import` statement
- Several built-in functions
  - `input`
  - conversions: `str`, `int`, `float`
  - `abs`, `min`, `max`, `pow`, and many more from the `math` library
  - `len`
  - `type`

SUMMARY (CONT'D)

- Binary operations (so far)
  - for integers: `+` `-` `*` `//` `%` `**`
  - for floats: `+` `-` `*` `/` `**`
  - for strings: `+` `*`

SUMMARY (CONT'D)

- Binary operations (so far)
  - for integers: `+` `-` `*` `//` `%` `**`
  - for floats: `+` `-` `*` `/` `**`
  - for strings: `+` `*`

SUMMARY (CONT'D)

- The Python interpreter can be run *interactively* or *not*.

  - When **interactive**, you type in a statement or an expression.
    - ➡ When a statement is entered, it gets executed.
      - ✦ If there is any output, it appears on subsequent lines.
    - ➡ When an expression is entered, it gets evaluated.
      - ✦ The value that results is displayed on the next line.

  - When **not interactive**, Python just loads and runs a script.
    - ➡ Its code is executed, line by line (statement followed by statement).

CONDITIONAL EXECUTION

LECTURE 01–2 PART 2

THE CONDITIONAL STATEMENT
THE BOOLEAN TYPE

JIM FIX, REED COLLEGE CSCI 121

"FLOW OF CONTROL"

NOTE: so far, Python programs perform "*straight-line*" execution.

- The interpreter goes through the code line-by-line, tracking where it's at with an instruction pointer.
  - ➡ The movement of that pointer is called the program's *flow of control*.

- With code that has ***conditional statements*** and ***loops***, we'll see flow that's not just top to bottom.
  - ➡ Lines might get *repeatedly executed*, or lines might get *skipped*.

- With conditions and loops, there are ***branches*** in the possible flows.

"BRANCHING"

- Here is an example of a conditional (or "if") statement:

```
pi = 3.14159
area = float(input("Circle area? "))
if area < 0.0:
    print("That's not a valid area.")
else:
    radius = (area / pi) ** 0.5
    print("That circle's radius is "+str(radius)+".")
```

- Depending on the value of **`area`**, either the first **`print`** or the second **`print`** will execute.
  - ➡ The other one will get skipped.

"LOOPING"

- Here is an example of a looping "while" statement:

```
pi = 3.14159
area = float(input("Circle area? “))
while area < 0.0:
    print(“That’s not a valid area.”)
    area = float(input(“Try again:”))
radius = (area / pi) ** 0.5
print("That circle’s radius is “+str(radius)+”.”)
```

- Because of that **`while`** statement, the re-prompting and re-input of an **`area`** with that second **`input`** can be repeatedly executed.
  - ➡ Lines 3 and 4 are repeated until the user enters a good **`area`** value.

THE "IF-ELSE" CONDITIONAL STATEMENT

- Python allows us to reason about values and act on them *conditionally*.
- For example, consider this script:

```
x = float(input("Enter a value: "))
if x < 0:
    abs_x = -x
else:
    abs_x = x
print("The absolute value of it is " + str(abs_x))
```

THE "IF-ELSE" CONDITIONAL STATEMENT

- Python allows us to reason about values and act on them *conditionally*.
- For example, consider this script:

```
x = float(input("Enter a value: "))
if x < 0:
    abs_x = -x
else:
    abs_x = x
print("The absolute value of it is " + str(abs_x))
```

```
% python3 absolute.py
Enter a value: -5.5
The absolute value of it is 5.5
% python3 absolute.py
Enter a value: 105.77
The absolute value of it is 105.77
% python3 absolute.py
Enter a value: 0.0
The absolute value of it is 0.0
```

THE "IF-ELSE" CONDITIONAL STATEMENT

- Python allows us to reason about values and act on them *conditionally*.
- For example, consider this script:

```
x = float(input("Enter a value: "))
if x < 0:
    abs_x = -x
else:
    abs_x = x
print("The absolute value of it is " + str(abs_x))
```

- When fed a negative value, it prints the value with its sign flipped.
  - ➡I.e. the positive value with the same magnitude. `-5.5` ~> `5.5`
- Otherwise, if positive or `0.0`, it just prints that value.

SYNTAX: IF–ELSE STATEMENT

Below is a template for conditional statements:

```
if condition-expression:
        lines of statements executed if the condition holds
        ...
else:
        lines of statements executed if the condition does not hold
        ...
lines of code executed after, in either case
```

CONDITION EXPRESSIONS COMPUTE A BOOL VALUE

```
>>> 345 < 10
False
>>> 345 == 300 + 50 - 5
True
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
>>> x = 57
>>> (x > 0) and (x <= 100)
True
>>> (x <= 0) or (x > 100)
False
>>> not (345 < 10)
True
>>> not ((x <= 0) or (x > 100))
True
```

CONDITIONAL STATEMENT EXECUTION

- Python allows us to reason about values and act on them *conditionally*.
- For example, consider this script:

```
x = float(input("Enter a value: "))
if x < 0:
    abs_x = -x
else:
    abs_x = x
print("The absolute value of it is " + str(abs_x))
```

When the script is run, the **`if`** code gets executed as follows:

- Python first checks the condition before the colon.
  - ➡ If the condition is **`True`**, it executes the first **`return`** statement.
  - ➡ If the condition is **`False`**, it executes the second **`return`** statement.
    This is the one sitting under the **`else`** line.

CONDITIONAL STATEMENT EXECUTION

- Python allows us to reason about values and act on them *conditionally*.
- For example, consider this script:

```
x = float(input("Enter a value: "))
if x < 0:
    abs_x = -x
else:
    abs_x = x
print("The absolute value of it is " + str(abs_x))
```

- You could maybe say that **`if-else`** gives Python code "intelligence."
  - ➡ It reasons about the value of **`x`** and behaves one way or the other.
- The code is smart!

SYNTAX: IF–ELSE STATEMENT

Below is a template for conditional statements:

```
if condition-expression:
    lines of statements executed if the condition holds
    ...
else:
    lines of statements executed if the condition does not hold
    ...
lines of code executed after, in either case
```

- Use indentation to indicate the "true" code block and the "false" code block.

ANOTHER EXAMPLE: CHECKING PARITY

- Here is a script that acts differently, depending on the *parity* of a number.

```
n = int("Enter an integer: ")
if n % 2 == 0:
    print(“That number is even.”)
else:
    print(“That number is odd.”)
```

- The equality test `==` is used to compare...
  - the left-hand expression's value `n % 2`
  - with the right-hand expression's value `0`.
- It is used to check whether they are equal.

CONDITIONAL STATEMENT WITH NO ELSE

- A different version of the absolute value script:

```
x = float(input("Enter a value: “))
if x < 0:
    x = -x
print("The absolute value of it is " + str(x))
```

SYNTAX: IF STATEMENT

Below is a template for conditional statements with no "else" block:

```
if condition-expression:
    lines of statements executed only if the condition holds
    ...
lines of code executed after, in either case
```

- Use indentation to indicate the "true" code block.

ANOTHER EXAMPLE: AUTOGRADER FEEDBACK

- The code below is like some code in some autograder:

```
print("Your code passed " + str(passed))
print(" out of " + str(tested) + "tests.")
if passed == tested:
    print("Your code passed all our tests!")
if not on_time:
    print("But you submitted after the deadline.")
```

NESTING CONDITIONAL STATEMENTS

- The code below is like some code in some autograder:

```
if on_time:

    if all_correct:
        mesg =  "Great work passing all the tests!\n"
        mesg += "You've earned the points for this problem."
    else:
        mesg = "To earn points, make sure all the tests pass."

else:

    if all_correct:
        mesg =  "Great work making all the tests pass.\n"
        mesg += "Sadly we can't offer you any points.\n"
        mesg += "You submitted this after the deadline."
    else:
        mesg =  "Sorry! There's still a problem. No points."

print(mesg)
```

NEXT TIME

- More examples using `if`.
- More about boolean expressions.
  - comparisons: `<` `<=` `>` `>=`
  - logical operations: `and` `or` `not`
  - storing boolean results
- Looping with `while`.