CS 115 Lecture
Conditionals and if statements

Taken from notes by Dr. Neil Moore

Selection

Sometimes we want to execute code only sometimes.

- "Run this code in a certain situation."
  - How do you express "in a certain situation" in code?
- "Run this code if this expression is true."
  - So we'll need expressions that can be True or False
  - We mentioned a type that has 2 values, True or False, the second week of class
  - `bool` (Booleans)

The boolean type

- The type `bool` in Python represents a value that is either true or false.
  - Two literals (constants): `True` and `False`
    - Case-sensitive as always!
  - You can have boolean variables:

    ```
    is_finished = False
    ```

    - Sometimes they are called **flags** (more later)
  - … and boolean expressions:

    ```
    is_smallest = number < minimum
    can_run = have_file and is_valid
    ```

Naming boolean variables

This isn't a hard-and-fast rule but try to name boolean variables as a sentence or sentence fragment:

- Is this item selected? `is_selected`
- Is the user a new user? `user_is_new` (or `is_user_new`)
- Does the program have an input file? `have_input_file`
- Does the user want the answer in meters? `want_meters`

Why `is_selected` and not just `selected`?

- Ambiguous: it could also mean "which item is selected?"

Equality

Other than literal `True` and `False`, the simplest boolean expressions compare the values of two expressions.

- Less than, greater than, …
- Even simpler: "is equal to" and "is not equal to"
  - The equal sign is already taken for assignment
  - So equality testing uses the symbol ==

```
logged_in = password == "hunter1"
```

    - No spaces between the two equal signs
- Contrast: "**==**" compares values, the "**is**" operator asks "are they aliases?" (names for the same object)
  - **is operator** will not be needed in this class

Inequality

- It’s kind of hard to type ≠ so Python uses != for the relationship “is not equal to”

```
need_plural = quantity != 1
did_fail = actual != expected
```

- How does an assignment statement like this work? Like any other!
  - Evaluate the right hand side (the != gives a True or False value)
  - Store that True or False value in the variable on the left hand side

Comparisons

Besides equality and inequality, Python has four more comparison (**relational)** operators:

- Less than (<) and greater than (>):

```
score < 60
damage > hit_points
```

- Less than or equal to (<=), greater than or equal to (>=)

```
students <= seats
score >= 60
```

Comparisons

- Precedence: all relational operators are lower than arithmetic operators, and are higher than the assignment operator and logical operators

```
need_alert = points + bonus < possible
* 0.60
```

is the same as

```
need_alert = ((points + bonus) <
(possible * 0.60))
```

- Relational operators are all of equal precedence to each other, so if you have more than one in an expression, they are evaluated left to right

Relational operators and types

- What type does the relational operators return? (that is, the result)
  - `bool`
- What types can be compared using relational operators?
  - Numbers: `ints` and `floats`
  - Bools: True and False
  - Strings

Comparisons

- Relational operators cannot mix strings and number values

  `3 < “Hello”`

  TypeError: unorderable types: int() < str()
- It’s ok to mix `ints` and `floats` though
- Be careful comparing `floats` to `floats` – they may NOT be equal though we think they are, especially if the float was generated by repeated arithmetic operators

Comparing strings

- What does it mean to compare two strings?
  - When computers were new, each hardware manufacturer had their own code of numbers to stand for alphabetic characters
  - Users didn’t care as long as they press ‘A’ on keyboard and got ‘A’ on screen
  - When people wanted to swap data between hardware brands, found out they needed a **Standard Code** for encoding characters
  - Some competition, but ASCII won out! Being used by microcomputers didn’t hurt ASCII’s popularity either.
  - ASCII (and its superset, Unicode) is used more today than any other character code by FAR, 90% of the computers in the world use it
  - ASCII – American Standard Code for Information Interchange, created in the 70’s, Unicode created later (90’s, 2000’s)

Comparing strings

- Each character is assigned a numeric value, those values are actually what’s compared
- Alphabetic characters are in alphabetic order
  - ‘A’ < ‘B’ < ‘C’ < … < ‘X’ < ‘Y’ < ‘Z’ (upper case)
  - ‘a’ < ‘b’ < ‘c’ < … < ‘x’ < ‘y’ < ‘z’ (lower case)
- Uppercase Z comes before lowercase a
  - ‘A’ < … < ‘Z’ < … < ‘a’ < … < ‘z’
- Digits are in numeric order ‘0’ < ‘1’ < ‘2’ < … < ‘9’
- Digits come before alphabetic characters
- And “ “ (one space) comes before all other printable characters
- ‘ ‘ < ‘0’ < … < ‘9’ < … < ‘A’ < … < ‘Z’ < … < ‘a’ < … < ‘z’

ASCII and Unicode

- http://unicode-table.com/en/#control-character
- ASCII table

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

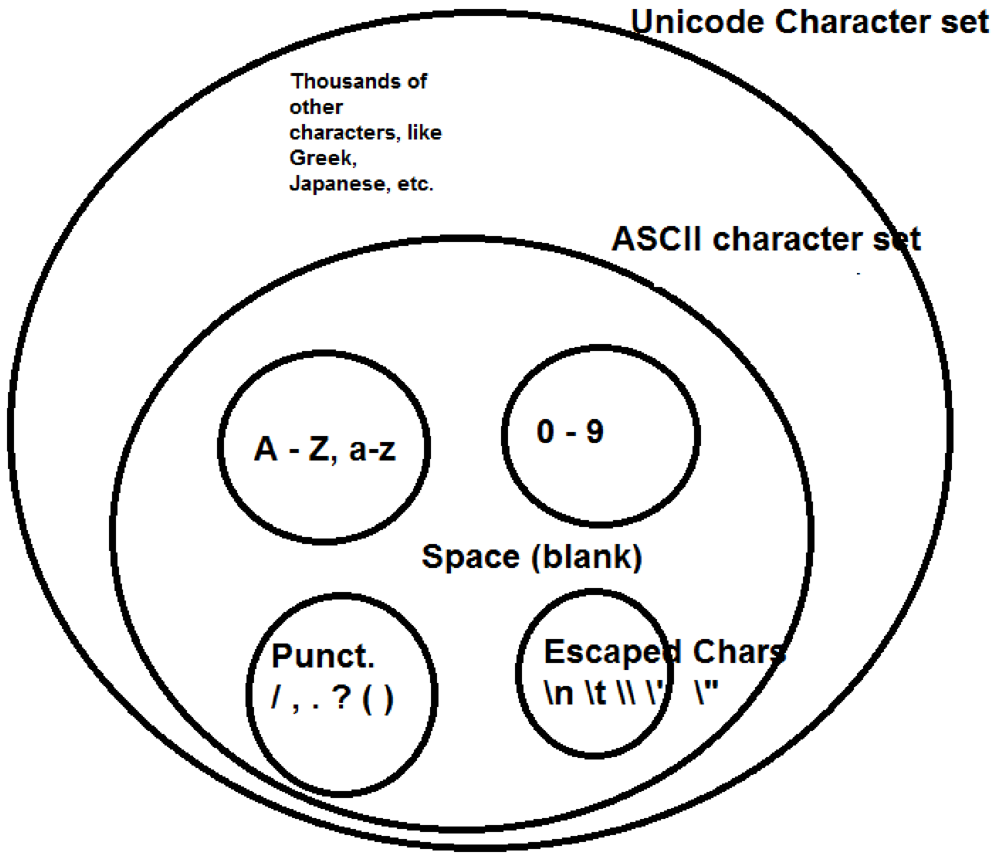
A note about “characters”

- Character vs. Letter
  - A Character is ANY symbol you can get from the keyboard (actually any element of the ASCII character code, or these days, of the Unicode character code)
  - A Letter is specifically an upper or lower case letter of the alphabet - a total of 52 characters

A note about “characters”

- Computer scientists use terms carefully
  - Number vs. numeric string
    - A number is a value that is stored in RAM where it can be accessed, in some pattern of bits. It has the data type of “integer” or “float” (in Python): examples: 4, 3.5, 9.e51
      - You use arithmetic operators with this data type
    - A numeric string is a string of characters which has only digits, a “+” or a “-” at the front, possibly a decimal point “.”, possibly an “e”, in the right order. It is a string, not an int or a float
      - You cannot use arithmetic operators with this type
  - Digits vs. Letters: ‘0’…’9’ vs. ‘A’…’Z’,’a’…’z’
    - There is no overlap between these two sets of characters

Relationship of groups of characters



Comparing strings

- Comparing single character strings means comparing their ASCII (Unicode) codes, but what if they are longer strings?
- The algorithm says
  - Start at characters at left end of each string
  - Compare – are they the same or different?
  - If they’re different, you can decide which is less and you’re done!
  - If they are the same, move one character to the right in each string and repeat comparison of characters
  - You’ll either run out of string or find a difference
  - If run out of both strings at same time, they’re equal
  - If run out of one before the other, the shorter is less

Chaining comparisons

- In Python, comparisons can be chained together:

  ```
  if 0 < x < y <= 100:
  ```

- It means: 0 < `x` and `x < y` and `y` <= 100
- This notation is common in mathematics
  - But not in most programming languages!
  - Python is rather unusual in allowing it

The if statement

Now that we can write some boolean statements, how do we use those to control whether or not certain statements execute?

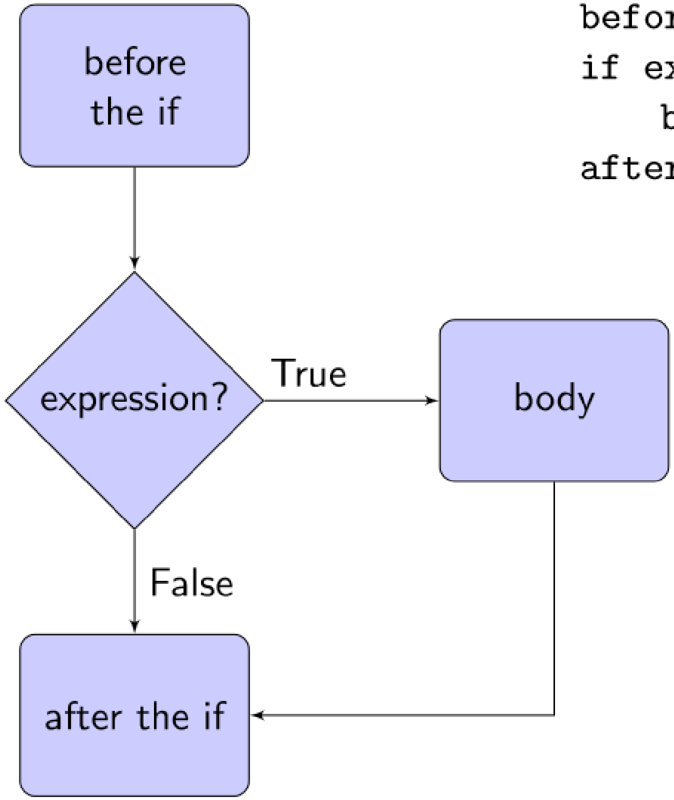
- Use an **if** statement
- Syntax

```
if expression:
    body
```

- The expression should evaluate to True or False
- The body is an indented block of code
- Semantics: 1. Evaluate the expression on the first line
  2. Runs the body if the expression was True
  3. Goes on to the code line after the body, either way

Flowchart for if

Flowchart for `if`




```
before the if
if expression:
    body
after the if
```

Alternatives: else

Commonly we want to do **either** this **or** that (but not both).

- In Python we can use an **else** block. Syntax:

```
if expression:
    if-body
else:
    else-body
```

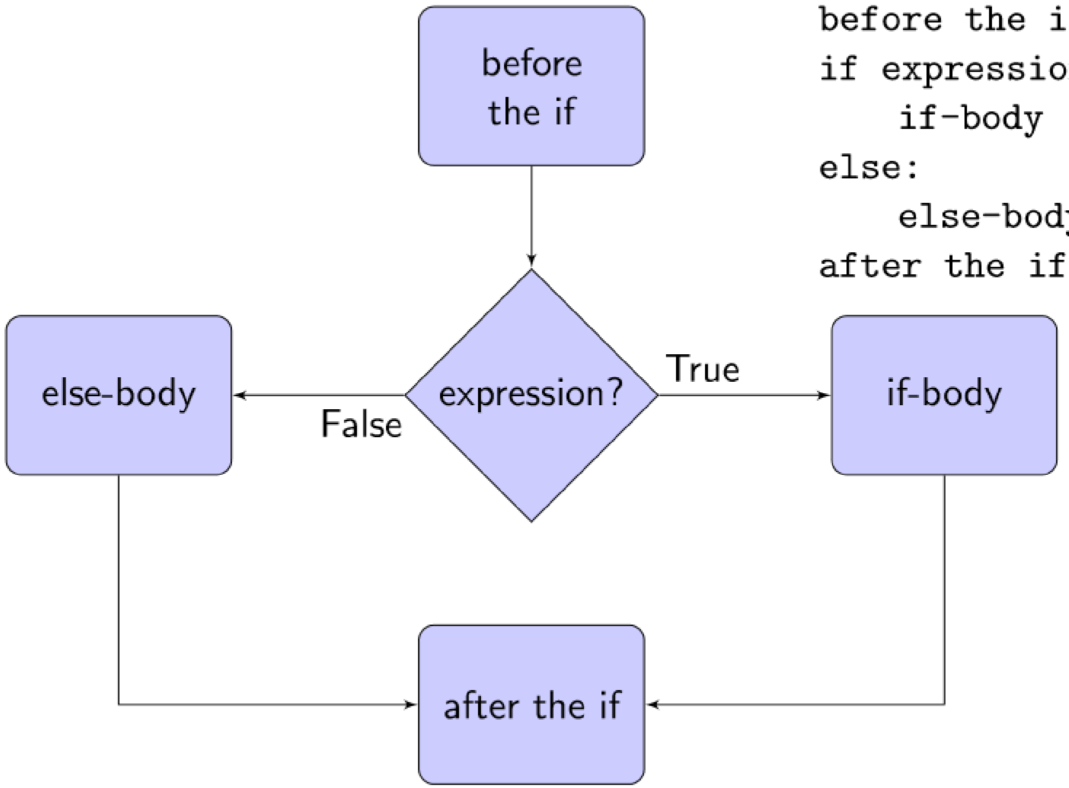
  - Both bodies are indented blocks
  - No expression on the line with else!
  - Cannot have an else without an if first!

Alternatives: else

- Semantics:
  - Always evaluates the expression on first line
  - If the expression is True, runs the if-body
  - If the expression is False, runs the else-body
  - Either way, goes on to the code line after the else-body
- Only use `else` if there is something to do in the False case
  - **It's ok not to have an else for an if!**

Flowchart for if-else

Flowchart for `if-else`




```
before the if
if expression:
    if-body
else:
    else-body
after the if
```

Many alternatives

Sometimes there are more than two alternatives

- Converting a numeric score into a letter grade:
  - If the score is greater than or equal to 90, print A
  - Otherwise, if score is greater than or equal to 80, print B
  - Otherwise, if score is greater than or equal to 70, print C
  - And so on…
- We want to run exactly **one** piece of that code
  - even though 95 >= 70, we don't want 95 to cause C to print too!
  - First check if score is >= 90
  - If that was False, check if score >= 80
  - If that is False too, check if score >= 70, …
- The order matters!
  - What would happen if we swapped the order of the B and C tests?
  - Then we'd never report a B!

Chained alternatives: elif

- Syntax:

```
if expression 1:
   body 1
elif expression 2:
   body 2
elif expression 3:
   body 3
…
```

- Each `elif` is followed by an expression and a colon
- Each body is an indented block
- You **can** have an else block at the very end. It is **not** required.

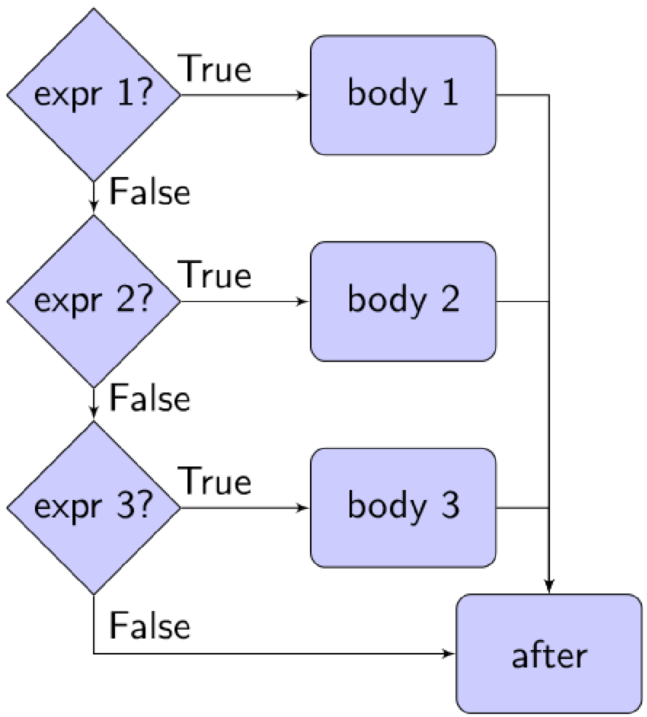
Chained alternatives: semantics

Semantics:

- Evaluates expression 1
- If expression 1 was True, run body 1 (and that’s ALL)
- If expression 1 was False, evaluate expression 2
- If expression 2 was True, run body 2 (and that’s ALL)
- If expression 2 was False, evaluates expression 3
- After running **at most one** body, goes on to the next line of code after the end of the chained if statement
- Only runs **one** body, or none
- It runs the body of the first True expression

Flowchart for if / elif

Flowchart for `if-elif`




```
if expr1:
    body1
elif expr2:
    body2
elif expr3:
    body3
after
```

Open and closed selection

- If there is an **else** in a chained if/elif, the selection is called **closed**
  - Meaning that exactly one of the bodies will run
- Otherwise it is **open**: zero or more bodies will be run
- If the last **elif** is supposed to cover all the remaining cases, you should prefer **else** instead:

```
if score >= 90:
        grade = ‘A’
elif score >= 80:
        grade = ‘B’
elif score >= 70:
        grade = ‘C’
elif score >= 60:
        grade = ‘D’
else:
        grade = ‘E’
```

Thinking about ifs

- How is the problem you are solving expressed?
  - Are you looking for specific values, nothing else is of interest?
  - Are you dividing a range up into specific segments, so that everything is of interest?
  - Are you testing for conditions that are mutually exclusive? (if one is True, the rest cannot be)
  - Or conditions that overlap? (that are independent of each other, if one is True, the others may or may not be)

Example for specific values

```
if x == 5:
   print(“do the 5 thing”)
elif x == 19:
   print(“do the 19 thing”)
elif x == -2:
   print(“do the -2 thing”)
```

nothing else to do here, if it’s not 5, 19 or -2 I don’t
#care

there is NO else here!

does order matter here?

Dividing a range

```
if x > 100:
    print("wonderful!")
elif x >= 70:
    print("ok")
elif x >= 55:
    print("meh")
else:
    print("bleah")
# need the last else to catch
#"everything lower"
#Note that order matters here!
```

Mutually exclusive conditions

```
if x % 10 == 5:
    print(“that’s a number ending in 5”)
else:
    print(“not a 5 on the end!”)
```

you do NOT need “elif x % 10 != 5:”

if the first test is False, the other must be True

Overlapping conditions

```
if x > 50:
    print("too high!")
if x % 2 == 0:
    print("that's even")
```

two separate, independent if's, because the

tests were independent of each other.

if one is True, the other may or may not be True

does the order of the if's matter?

Using elif

If you want more than one branch to execute, you don’t want **elif**. There you would use a sequence of separate if statements

“Factoring out” code

- If you write an if with more than one branch, look carefully at the code in the branches. If the SAME statement appears in both the branches (for an if/else) or in ALL the branches for a multiway if/elif/else statement, see if you can “factor it out”.
- Example:

```
if z > b:
        print(z)
        z = b * 2
else:
        print(z)
        z = 9 * b
```

- The code that is common to both branches should be done OUTSIDE the if statement altogether. In this case, do it before the if statement.

“Factoring out” code

- Why is this a big deal?
  - Efficiency – why write code twice?
  - Less code to debug
  - More likely to get it right if it only appears in ONE place instead of 2 or 3!
- Be careful! It may not always be possible. In this example, If the print in the else-block had come AFTER the assignment statement, then the prints would have been doing different things and should NOT be factored out

Testing if statements

When testing programs with if statements, be sure to consider and test **all** the possible outcomes.

- If your tests never execute a particular line of code, you don't know if it works!
- For every if or if-else, you should have two cases:
  - one where the test is True
  - one where the test is False – even if there is no explicit else branch.
- For a chained if/elif, you should test
  - Expression 1 is True
  - Expression 1 is False, expression 2 is True
  - Expression 1 and 2 are False, expression 3 is True
  - …
  - All the expressions are False
  - If you had a chain with N elif's, you should have N+2 test cases

More testing

- It helps to consider combinations of separate if statements too
  - Especially when they use the same variables

```
if userID != “admin”:
     is_valid = False
if password != “password”:
     is_valid = False
```

- You can get four test cases for these two if’s
  - UserID right, password right
  - UserID right, password wrong
  - UserID wrong, password right
  - UserID wrong, password wrong

More testing

- Don’t forget to check the **boundary cases**
  - what if the score is exactly 60.0?
  - what if the score is 59.9?
  - what if the score is 0?
  - what if the score is 101?