Introduction to Python Programming

General Information

- Unlike C/C++ or Java, Python statements do not end in a semicolon
- Whitespace is meaningful in Python: especially indentation and placement of newlines.
- In Python, indentation is the way you indicate the scope of a conditional, function, etc.
- Look, no braces!
- Python is interpretive, meaning you don't have to write programs.
- You can just enter statements into the Python environment and they'll execute

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!")
```

The Python Shell

- Because Python is interpretive, you can do simple things with the shell
- At the prompt, type `python`
- You should have a >>> prompt
- Type in:

```
print("hello, world")
```

- You have written your first Python program
- Save it as as HelloWorld.py
- You **must** provide the .py extension
- `Type python HelloWorld.py`
- **Note that Linux is case sensitive**

Python and type

Python determines the data types in a program automatically. "Dynamic Typing"

- That is, you don't declare variables to be a specific type
- A variable has the type that corresponds to the value you assign to it

But Python's not casual about types, it enforces them after it figures them out. "Strong Typing"

So, for example, you can't just append an integer to a string. You must first convert the integer to a string itself.

```
>>>
x = "the answer is "  # Decides x is string.
y = 23                # Decides y is integer.
print x + y   # Python will complain about this.
type(x)
type(y)
```

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: `input("")`

- 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)
```

Variables

- As in every language, a variable is the name of a memory location

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.

  bob Bob _bob _2_bob_ bob_2 BoB

- There are some reserved words which can not be used as variables:

  ```
  and, assert, break, class, continue, def,
  del, elif, else, except, exec, finally, for,
  from, global, if, import, in, is, lambda,
  not, or, pass, print, raise, return, try,
  while
  ```

Variables

- No need to declare
- Need to assign (initialize)
  - use of uninitialized variable raises exception
- Not typed

```
if friendly: greeting = "hello world"
else: greeting = 12**2
print greeting
```

- If you try to access a name before it's been properly created (by placing it on the left side of an assignment), you'll get an `NameError` error.

Variables

- You can also assign to multiple names at the same time.

```
>>> x, y = 2, 3
>>> x
2
>>> y
3
```

Variables

Assignment manipulates references

x = y **does not make a copy** of y

x = y makes x **reference** the object y references

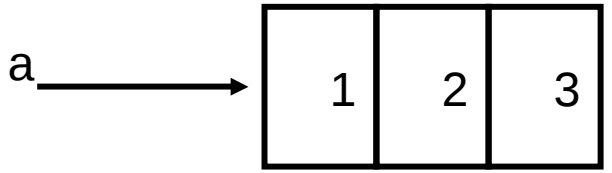
Very useful; but beware!

Example:

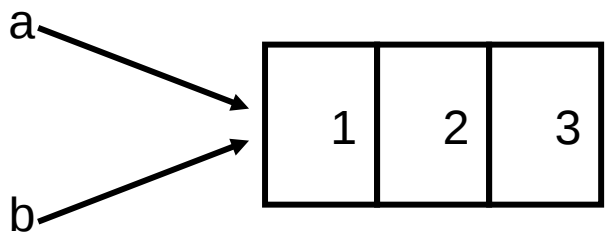
```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> print b
[1, 2, 3, 4]
```

Changing a shared list

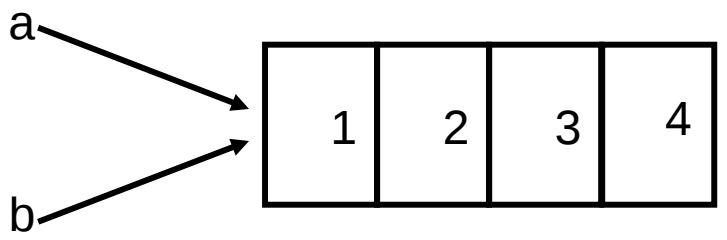
a = [1, 2, 3]



b = a



a.append(4)

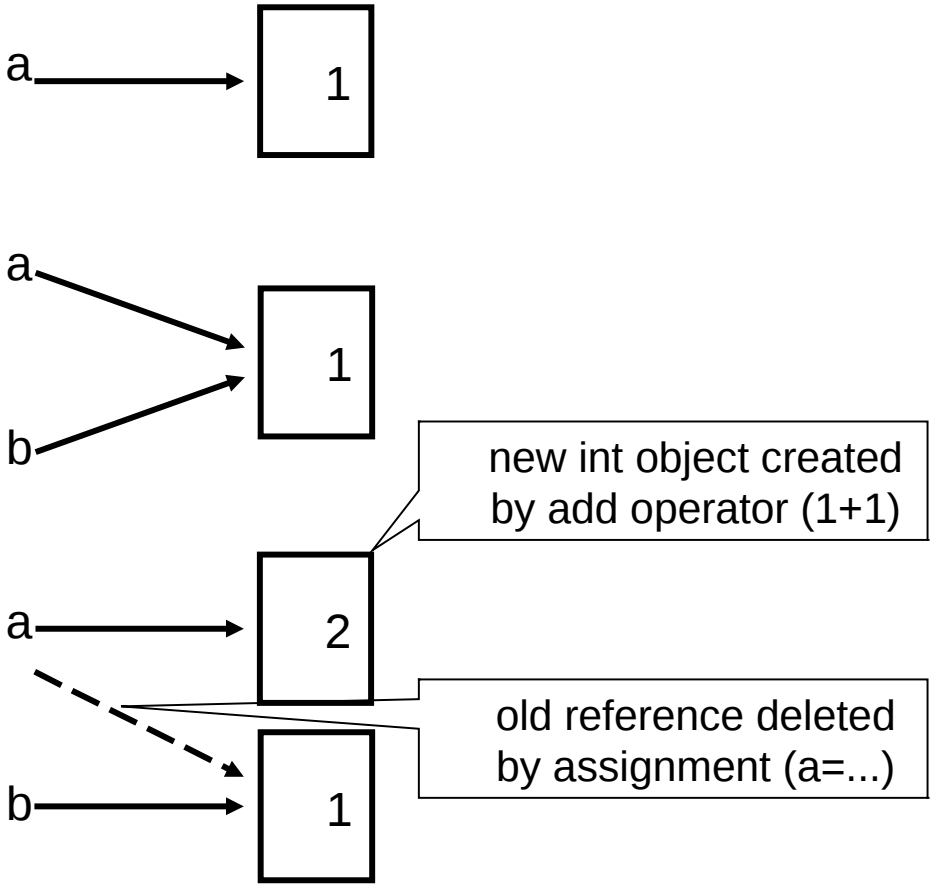


Changing an integer

a = 1

b = a

a = a+1



Comments

- All code must contain comments that describe what it does
- In Python, lines beginning with a # sign are comment lines

```
# This entire line is a comment
x=5   # Set up loop counter
```

Operators

- Arithmetic operators we will use:

  `+ - * /` addition, subtraction/negation, multiplication, division

  `%` modulus, a.k.a. remainder

  `**` exponentiation

- **precedence**: Order in which operations are computed.
  - `* / % **` have a higher precedence than `+ -`

    `1 + 3 * 4` is 13

  - Parentheses can be used to force a certain order of evaluation.

    `(1 + 3) * 4` is 16

Expressions

- When integers and reals are mixed, the result is a real number.
  - Example: `1 / 2.0` is `0.5`

  - The conversion occurs on a per-operator basis.
  - Integer division truncates :-(

```
7 / 3 * 1.2 + 3 / 2
  2   * 1.2 + 3 / 2
    2.4     + 3 / 2
    2.4     +   1
        3.4
```

Math Functions

- Use this at the top of your program: `from math import *`

Command name	Description
`abs(`***value***`)`	absolute value
`ceil(`***value***`)`	rounds up
`cos(`***value***`)`	cosine, in radians
`floor(`***value***`)`	rounds down
`log(`***value***`)`	logarithm, base *e*
`log10(`***value***`)`	logarithm, base 10
`max(`***value1***`, `***value2***`)`	larger of two values
`min(`***value1***`, `***value2***`)`	smaller of two values
`round(`***value***`)`	nearest whole number
`sin(`***value***`)`	sine, in radians
`sqrt(`***value***`)`	square root

Constant	Description
`e`	2.7182818...
`pi`	3.1415926...

Relational Operators

- Many logical expressions use *relational operators*:

Operator	Meaning	Example	Result
==	equals	`1 + 1 == 2`	`True`
!=	does not equal	`3.2 != 2.5`	`True`
<	less than	`10 < 5`	`False`
>	greater than	`10 > 5`	`True`
<=	less than or equal to	`126 <= 100`	`False`
>=	greater than or equal to	`5.0 >= 5.0`	`True`

Logical Operators

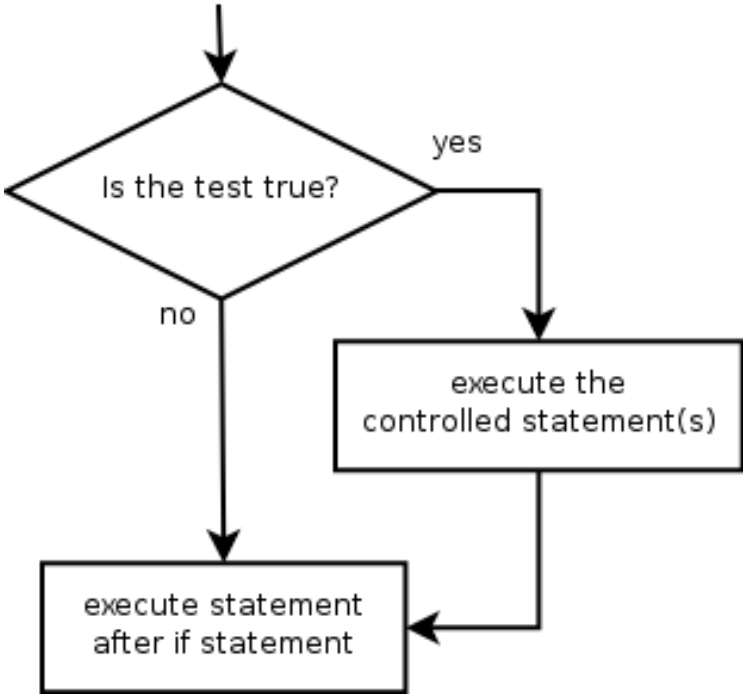
- These operators return true or false

Operator	Example	Result
and	9 != 6 and 2 < 3	True
or	2 == 3 or -1 < 5	True
not	not 7 > 0	False

The `if` Statement

- Syntax:

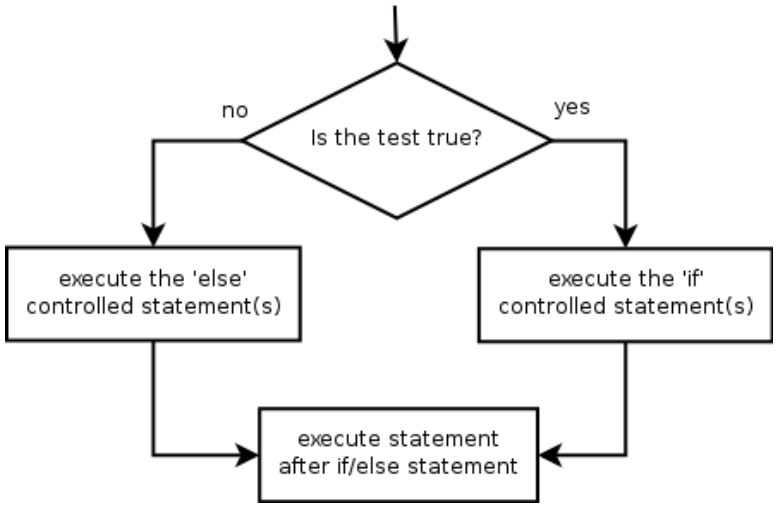
```
if <condition>:
   <statements>
```



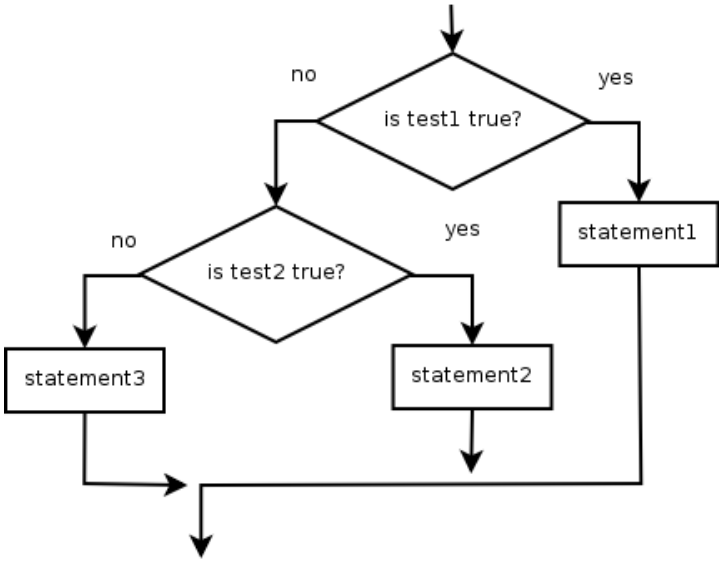

```
x = 5
if x > 4:
   print(“x is greater than 4”)
print(“This is not in the scope of the
if”)
```

The if/else Statement

```
if <condition>:
    <statements>
else:
    <statements>
```




```
if <condition>:
    statements
[elif condition:
    statements] ...
else:
    statements
```



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`

The for Loop

- This is similar to what you're used to from C or Java, but not the same
- Syntax:

```
for variableName in groupOfValues:
    <statements>
```

- variableName gives a name to each value, so you can refer to it in the statements.
- groupOfValues can be a range of integers, specified with the range function.

Example:

```
for x in range(1, 6):
    print x, "squared is", x * x
```

Output:

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
```

Range

- The range function specifies a range of integers:

`range(start, stop)` - the integers between start (inclusive) and stop (exclusive)

- It can also accept a third value specifying the change between values.

`range(start, stop, step)` - the integers between start (inclusive) and stop (exclusive) by step

Example:

```
for x in range(5, 0, -1):
    print x
print "Blastoff!"
```

Output:

```
5
4
3
2
1
Blastoff!
```

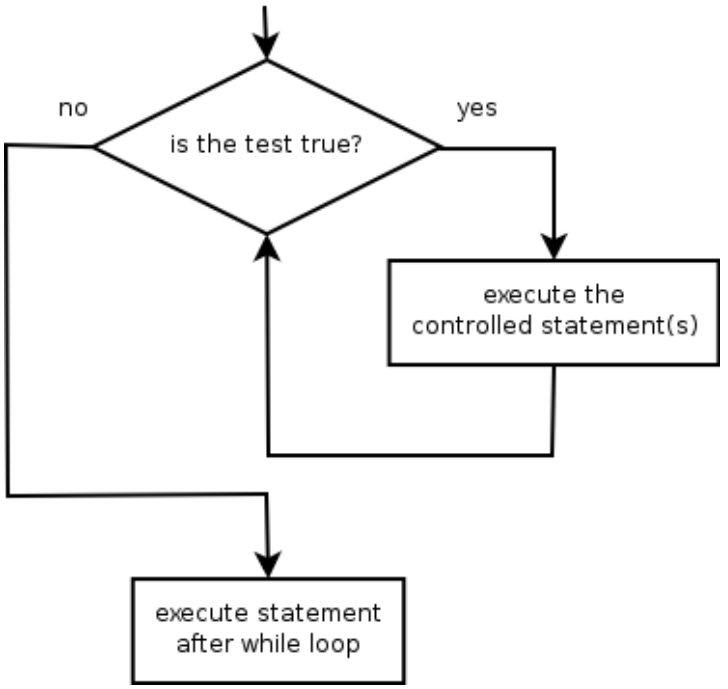
The while Loop

- Executes a group of statements as long as a condition is True.
- Good for indefinite loops (repeat an unknown number of times)
- Syntax:

```
while <condition>:
    <statements>
```

- Example:

```
number = 1
while number < 200:
    print number,
    number = number * 2
```



Grouping Indentation

In Python:

```
for i in range(20):
    if i%3 == 0:
        print i
        if i%5 == 0:
            print "Bingo!"
    print "---"
```

In C:

```
for (i = 0; i < 20; i++)
{
    if (i%3 == 0) {
        printf("%d\n", i);
        if (i%5 == 0) {
            printf("Bingo!\n"); }
    }
    printf("---\n");
}
```

```
0
Bingo!
---
---
---
3
---
---
---
6
---
---
---
9
---
---
---
12
---
---
---
15
Bingo!
---
---
---
18
---
---
```

`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>
```

Strings

- String: A sequence of text characters in a program.
- Strings start and end with quotation mark " or apostrophe ' characters.
- Examples:
  "hello"
  "This is a string"
  "This, too, is a string. It can be very long!"
- A string may not span across multiple lines or contain a " character.
  "This is not
  a legal String."

  "This is not a "legal" String either."

Strings-operators

- "hello"+"world" "helloworld" # concatenation
- "hello"*3 "hellohellohello" # repetition
- "hello"[0] "h" # indexing
- "hello"[-1] "o" # (from end)
- "hello"[1:4] "ell" # slicing
- len("hello") 5 # size
- "hello" < "jello" 1 # comparison
- "e" in "hello" 1 # search
- escapes: \n \t etc.

'single quotes' "double quotes" """triple quotes""" r"raw strings"

String Functions

- `len(`***`string`***`)` - number of characters in a string
- `str.lower(`***`string`***`)` - lowercase version of a string
- `str.upper(`***`string`***`)` - uppercase version of a string
- `str.isalpha(`***`string`***`)` - True if the string has only alpha chars
- Many others: split, replace, find, format, etc.
  - >>> "hello..{0}".format("world")
  - 'hello..world'

Other Built-in Types

- tuples, lists, sets, and dictionaries
- They all allow you to group more than one item of data together under one name
- You can also search them

Tuples

- Unchanging Sequences of Data
- Enclosed in parentheses:

```
tuple1 = (“This”, “is”, “a”, “tuple”)
print(tuple1)
```

- This prints the tuple exactly as shown

```
Print(tuple1[1])
```

- Prints “is” (without the quotes)

Tuples

key = (lastname, firstname)

point = x, y, z # parentheses optional

x, y, z = point # unpack

lastname = key[0]

singleton = (1,) # trailing comma!!!

empty = () # parentheses!

tuples vs. lists->tuples are immutable

Lists

- Changeable sequences of data

Lists are created by using square brackets:

```
breakfast = [ "coffee", "tea", "toast",
"egg" ]
```

- You can add to a list:

```
breakfast.append("waffles")
breakfast.extend(["cereal", "juice"])
```

- Same operators as for strings

a+b, a*3, a[0], a[-1], a[1:], len(a)

Lists

```
>>> a = range(5)                # [0,1,2,3,4]
>>> a.append(5)                 # [0,1,2,3,4,5]
>>> a.pop()                     # [0,1,2,3,4]
5
>>> a.insert(0, 42)             # [42,0,1,2,3,4]
>>> a.pop(0)                    # [0,1,2,3,4]
42
>>> a.reverse()                 # [4,3,2,1,0]
>>> a.sort()                    # [0,1,2,3,4]
```

Dictionaries

- Groupings of Data Indexed by Name
- Dictionaries are created using braces

```
sales = {}
sales["January"] = 10000
sales["February"] = 17000
sales["March"] = 16500
```

- The keys method of a dictionary gets you all of the keys as a list
- Hash tables, "associative arrays"

```
{"January":10000, "February":17000,"March":16500}
```

- Lookup :

```
sales["March"]
```

Dictionaries

- Keys, values, items:

sales.keys() -> [“January,“February”,“March”]

sales.values() -> [10000,17000,16500]

sales.items() ->

[(“January”,10000),(“February”,17000),(“March”:16500)]

- Presence check:

```
sales.has_key("January") ->
1
sales.has_key("spam") ->
0
```

- Values of any type; keys almost any
  - {"name":"Guido", "age":43, ("hello","world"):1, 42:"yes", "flag": ["red","white","blue"]}

Dictionaries

- Keys must be **immutable:**
  - numbers, strings, tuples of immutables
    - these cannot be changed after creation
  - reason is *hashing* (fast lookup technique)
  - **not** lists or other dictionaries
    - these types of objects can be changed "in place"
  - no restrictions on values
- Keys will be listed in **arbitrary order**
  - again, because of hashing

Sets

- Sets are similar to dictionaries in Python, except that they consist of only keys with no associated values.
- Essentially, they are a collection of data with no duplicates.
- They are very useful when it comes to removing duplicate data from data collections.
- Can do set operations – union(|),intersection(&), difference(-) and symmetrical difference (^), x in set

```
>>> s = {1,2,3,2,1}
>>> s
set([1, 2, 3])
>>> s = set([1,2,2,3,3,1,1])
>>> s
set([1, 2, 3])
```

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"`

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

STRINGS

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

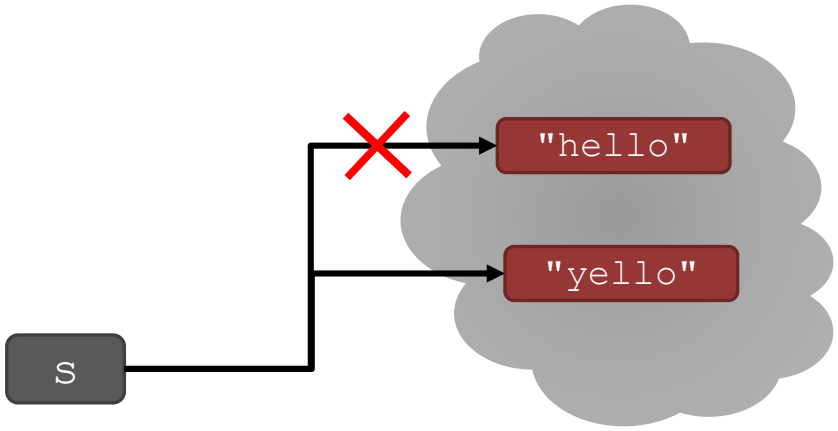
```
s = "hello"
```

```
s[0] = 'y'
```

→ gives an error

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

→ is allowed,
s bound to new object



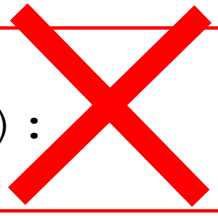
STRINGS AND LOOPS

- these two code snippets do the same thing
- bottom one is more “pythonic”

```
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")
```

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
```