Python Introduction

Python is an extremely popular computer language. Python is general purpose, good for solving many types of problems.

Compared to other languages Python is in the "programmer efficient" niche - allowing the programmer to specify what they want pretty easily, perhaps at the cost of running a little slower or using more memory.

The style of Python is minimal - don't require the programmer to type in a lot words. Mostly get out of the programmer's way. Python is designed to have a consistent style across the language. There is a thematic consistency in how its parts fit together across many situations.

Open Source

Python is distributed for free as open source software. This means many people and organization contribute code to make Python work, and it is free for anyone to use.

The central web site for Python information is python.org. Python was created in 1991 and for many years was run by Guido Van Rossum, who is kind of a rock star for creating such a world-influencing technology.

Cross Platform

Python is cross-platform, meaning a python program developed on Mac OS X, can likely also run on Windows or Linux without any change to the code. When the Python code gets to the part where it, say, it wants to check the mouse location, running on Windows, it uses the Windows-specific mouse facility, and running on the Mac it uses the Mac-specific facility. In this way, the programmer is insulated from many platform-specific details and their code just works.

Python Version 3

We will use Python version 3 in this course. Python version 3 made some changes vs. Python 2.x, in particular changing strings to be unicode, and the print() function to be a function. For the most part, Python 2.x code looks very similar to Python 3.x code. So working in both versions is not a big issue. That said, use of Python 2.x in the world is now dying out.

Python Interpreter

You write your Python code in a text file with a name like `hello.py`. How does that code Run? There is program installed on your computer named "python3" or "python", and its job is looking at and running your Python code. This type of program is called an "interpreter".

Talk To The Interpreter

A nice benefit of using an interpreter is that you can start interactive version of the interpreter and type Python code right into it to see what it does. What happens if you use `+` in Python code like this: `'hello' + 3`?

You could look up the answer to this, but a very quick way to find out is to type code into the interpreter and see what it does.

There are 2 ways to get the interpreter:

1. Open a command-line terminal. Mac: run the "Terminal" app in the Utilities folder. Windows: type "powershell" in the lower left, this opens the Windows command line terminal. In the terminal type the command "python3" ("python" on Windows). This runs the interpreter program directly. Type ctrl-d to exit when done (on Windows ctrl-z).

2. If you have PyCharm installed, at the lower-left of any window, click the Python Console tab. Or use the Tools > Python Console menu item.

Working With The Interpreter

The interpreter prints a ">>>" prompt and waits for you to type some Python code. It reads what you type, evaluates it, and prints the result - the so called read-eval-print loop (here what the user types is shown in bold)

```
>>> 1 + 2 * 3
7
>>> 'hello' + 'there'
'hellothere'
>>> max(1, 5, -2)
5
>>> 'hello' + 2
TypeError: can only concatenate str (not "int") to str
```

This is an excellent way to experiment with Python features to see how they work. If you have some code you are thinking about, it can be quick and informative to fire up the interpreter, and type the code in to see what it does.

Indent In Interpreter

It's possible to write multi-line indented code in the interpreter (we will cover those later in this reader). If a line is not finished, such as ending with ":", hitting return does not run the code right away. You can write further indented lines. When you enter a blank line, the interpreter runs the whole thing. (For throw-away code like this, I am willing to indent just 2 spaces).

```
>>> for i in range(10):
...   mult = i * 100
...   print(i, mult)
...
0 0
1 100
2 200
3 300
4 400
5 500
6 600
7 700
8 800
9 900
```

Interpreter Hacks: dir help

With the special **help** function, you can pull up the PyDoc for each function (the triple-quote comments for a function). To refer to a function within the `str` type use the form `str.find`. Note that the function name is **not** followed by parenthesis - this is the unusual case of referring to a function but not calling it.

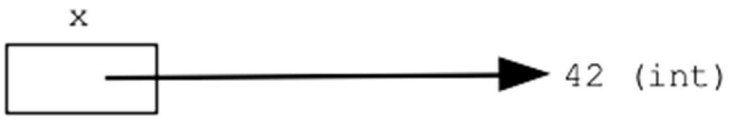
```
>>> help(str.find)
find(...)
    S.find(sub[, start[, end]]) -> int

    Return the lowest index in S where substring sub is found,
    such that sub is contained within S[start:end].  Optional
    arguments start and end are interpreted as in slice notation.

    Return -1 on failure.
```

Variables

A Python variable represents a little bit of memory, keeping track of a value as the code runs.

A variable is created simply with an "assignment" equal sign `=` stores a pointer to that value into the variable like this:

```
x = 42
```



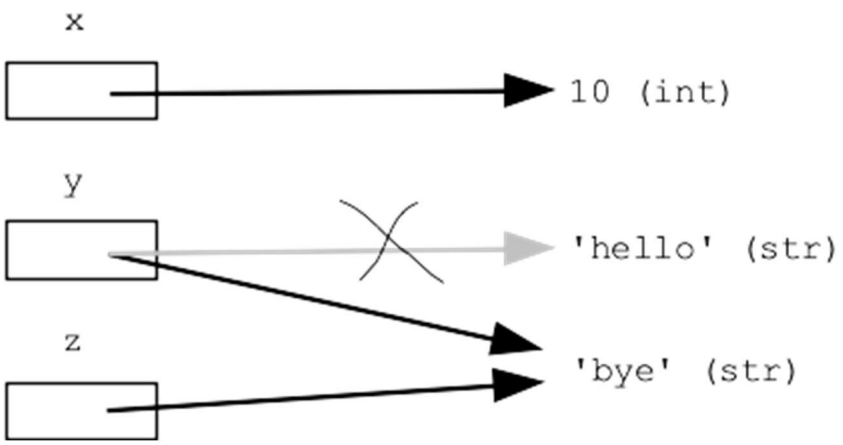
Later in the code, appearances of that variable name, e.g. `x`, retrieve its current value, in this case 42.

Trying to retrieve the value of a variable that does not exist fails with an error (i.e. no `=` ever assigned that variable).

Variable Assignment Rules

Here is a more complicated code example and a picture of memory for it, with the variables on the left and their values on the right.

```
x = 10
y = 'hello'
y = 'bye'
z = y
```



Things to notice here...

1. Assignment like `x = 10` above sets a pointer into that variable, pointing to the value.

2. Assignment like `y = 'bye'` above, where the variable has an existing pointer in it, overwrites the existing pointer with the new one. Put another way, a variable holds just one pointer. Assigning a new pointer gets rid of the old one.

3. Assignment between two variables like `z = y` above, makes them point to the same thing. It does not set up a permanent link between the variables; it is simpler than that. This assignment just changes `z` to point to the `y` value at that moment.

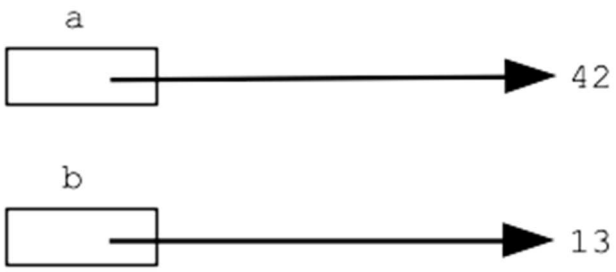
4. Small observation 1: each value in Python is tagged with its type, so in the example above the number 10 is tagged with the type "int", and the string `'hello'` is tagged with its type "str".

Garbage Collection Aside: in the above example, the string 'hello' exists in memory at one time, but then the variable `y` is moved to point to something else. When a value such as a 'hello' here is left with no variable pointing to it, it cannot be used any more by the code. Such memory, storing a value that is not used, is called "garbage", and the "garbage collector" facility reclaims that memory, re-using it to hold new values. This is something Python does automatically behind the scenes. Most modern languages have a garbage collection to reclaim memory automatically.

Variable Swap

Suppose we have two variables and we want to "swap" their values, so each takes on the value of the other. This is a little coding move that all programmers should know.

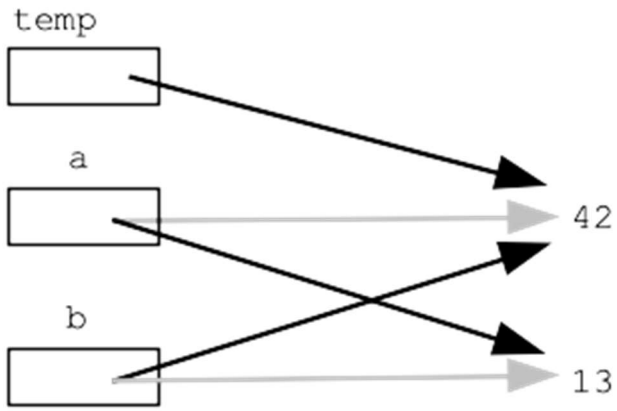
```
a = 42
b = 13
```



It might seem that one can begin with `a = b`, but this does not work, since it overwrites and thus loses the original value of `a`. The classic 3-line solution uses a temporary variable named "temp" to hold this value during the swap, like this:

```
temp = a
a = b
b = temp
```

Starting with the above diagram, you can trace through the three assignments, leading to this memory structure:



Assignment = is Shallow

The swap example demonstrates a key feature of Python assignment — each assignment `=` merely changes what a variable points to. The int, string etc.

values are undisturbed. The assignment = in Python just moves an arrow around. In the diagram, this changes the arrows on the left, but the values on the right are undisturbed.

We'll revisit this shallow quality of assignment with parameters and nested data structures.

Variable Names are Superficial Labels

Normally variables names are chosen to reflect what data they contain. That said, there is one funny feature of variable names to know.

Consider the following computation

```
>>> x = 6
>>> y = x + x
>>> y
12
```

Using a couple variables, it computes that doubling 6 makes 12. Suppose instead it was written this way:

```
>>> alice = 6
>>> bob = alice + alice
>>> bob
12
```

This is **exactly** the same computation, just using different variable names. What matters in a computation is the structure — which value is used at each spot in the computation. The variable names are just labels on the computation. If we change a variable name consistently throughout the code, the computation will work the same.

That said, though variable names are superficial, good code uses meaningful variable names, reflecting the role of that data in the algorithm.

Python Math

Numbers - int and float

Surprisingly, there are two distinct types of numbers for doing arithmetic in a computer - "int" for whole integer numbers like 6 and 42 and -3, and "float" for numbers like 3.14 with a decimal fraction.

Int Type

The Python "int" type represents whole integer values like 12 and -2. Addition, subtraction, and multiplication and division work the usual way with the operators: `+ - * /`. Division by zero is an error.

```
>>> 1 + 10 - 2
9
>>> 2 * 3 * 4
24
>>> 2 * 6 / 3
4.0
>>> 6 / 0
ZeroDivisionError: division by zero
```

Precedence

Just as in regular mathematics, multiplication and division have higher "precedence" than addition and subtraction, so they are evaluated first in an expression. After accounting for precedence, the arithmetic is done left-to-right.

e.g. here the multiplication happens first, then the addition:

```
>>> 1 + 2 * 3
7
>>> 1 + 3 * 3 + 1
11
```

Add parenthesis into the code to control which operations are evaluated first:

```
>>> (1 + 2) * 3
9
```

60 / 2 * 3

What is the value of `60 / 2 * 3'?

The evaluation proceeds left-to-right, applying each operator to a running result which is simple but can be unintuitive. For `60 / 2 * 3`, the steps are..

```
1. start with 60
2. 60 / 2 yielding 30.0
```

```
3. 30.0 * 3 yielding 90.0
```

The 2 is in the denominator, but the 3 is not. Add parenthesis to put both numbers in the denominator e.g. `60 / (2 * 3)`

```
>>> 60 / 2 * 3
90.0
>>> 60 / (2 * 3)
10.0
```

Division / Yields Float

One problem problem with `/` is that it does not produce an int, it produces a float. This is basically reasonable — 7 divided by 2 isn't an integer.

```
>>> 7 / 2
3.5        # a float, notice the "."
```

Adding subtracting or multiplying two ints always yields an int result, but division is different. The result of division is always a float value, even if the division comes out even.

```
>>> 9 / 2
4.5
>>> 8 / 2
4.0
>>> 101 / 8
12.625
```

// int Division

Many times an algorithm makes the most sense if all of the values are kept as ints, so we need an alternative to the `/` which produces floats. In Python the int-division operator `//` rounds down any fraction, always yielding an int result.

```
>>> 9 / 2    # "/" yields a float, not what we wanted
4.5
>>> 9 // 2   # "//" rounds down to int
4
>>> 8 // 2
4
>>> 87 // 8
10
```

** Exponentiation

The `**` operator does exponentiation, e.g. `3 ** 2` is 3^2

```
>>> 3 ** 2
9
>>> 2 ** 10
```

```
1024
```

Unlike most programming languages, Python int values do not have a maximum. Python allocates more and more bytes to store the int as it gets larger. The number of grains of sand making up the universe when I was in school was thought to be about 2^{100}, playing the role of handy very-large-number (I think it's bigger now as they keep finding more universe, but this number is handy). In Python, we can write an expression with that number and it just works.

```
>>> 2 ** 100
1267650600228229401496703205376
>>> 2 ** 100 + 1
1267650600228229401496703205377
```

Memory use approximation: int values of 256 or less are stored in a special way that uses very few bytes. Other ints take up about 24 bytes each in RAM.

Int Mod %

The "modulo" or "mod" operator `%` is essentially the remainder after division. So `(23 % 10)` yields 3 — divide 23 by 10 and 3 is the leftover remainder.

```
>>> 23 % 10
3
>>> 36 % 10
6
>>> 43 % 10
3
>>  40 % 10  # mod result 0 = divides evenly
0
>>> 17 % 5
2
>>> 15 % 5
0
```

If the modulo result is 0, it means the division can out evenly, e.g. `40 % 10` above. The best practice is to only use mod with non-negative numbers. Modding by 0 is an error, just like dividing by 0.

```
>>> 43 % 0
ZeroDivisionError: integer division or modulo by zero
```

Review Expressions

What is the value of each expression? Write the result as int (6) or float (6.0).

```
>>> 2 * 1 + 6
```

```
??
>>> 20 / 4 + 1
??
>>> 20 / (4 + 1)
??
>> 40 / 2 * 2
??
>>> 5 ** 2
??
>>> 7 / 2
??
>>> 7 // 2
??
>>> 13 % 10
??
>>> 20 % 10
??
>>> 42 % 20
??
>>> 31 % 20
??
```
Show

Float Type

Floating point numbers are used to do math with real quantities, such as a velocity or angle. The regular math operators `+ - * / **` all work with floats, producing a float result. If an expression mixes some int values and some float values, the math is converted to float - a one-way street called "promotion" to float.

```
>>> 1.0 + 2.0 * 3.0
7.0
>>>
>>> 1 + 2 * 3.0
7.0
>>>
>>> 2.1 ** 2
4.41
```

Float values can be written in scientific notation with the letter 'e' or 'E', like this:

```
>>> 1.2e23 * 10
1.2e+24
>>> 1.0e-4
0.0001
```

Float Error Term

Famously, floating point numbers have a tiny error term that builds up way off 15 digits or so to the right. Mostly this error is not shown when the float value is printed, as a few digits are not printed. However, the error digits are real, throwing the float value off a tiny amount. The error term will appear sometimes, just as a quirk of how many digits are printed (see below). This error term is an intrinsic limitation of floating point values in the computer. (Perhaps also why CS people are drawn to do their algorithms with int.)

```
>>> 0.1 + 0.1
0.2
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
0.7
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
0.7999999999999999
```

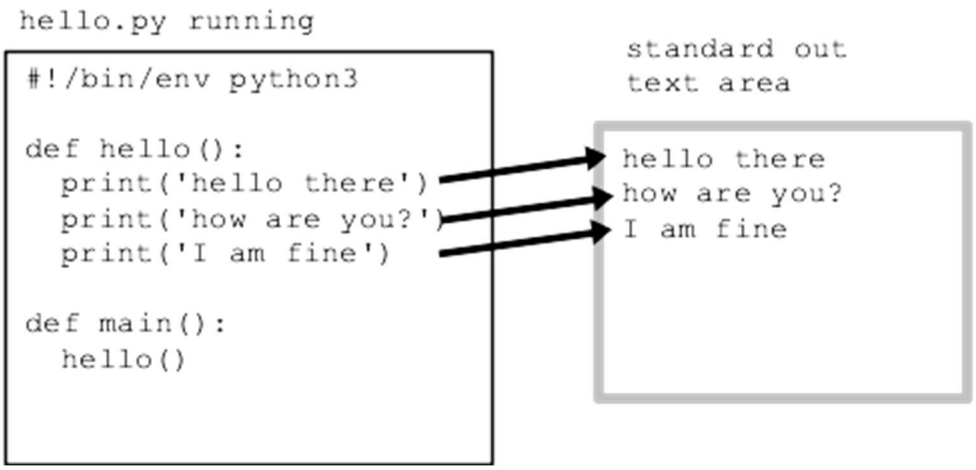
Mostly, an error 15 or so digits off to the right does not invalidate your computation. However, it means that code should not use `==` with float values, since the comparison will be thrown off by the error term. To compare float values, subtract them and compare the absolute value of the difference to some small delta.

```
>>> # are float values a and b the same?
>>> diff = abs(a - b)   # abs() is absolute value
>>> if diff < 1.0e-9:   # if diff less than 1 billionth,
    ...
```

Memory use approximation: float values take up about 24 bytes apiece.

`print()` and Standard Out

Every running program has a text output area called "standard out", or sometimes just "stdout". The Python print() function takes in python data such as ints and strings, and prints those values to standard out.



To say that standard out is "text" here means a series of lines, where each line is a series of chars with a `'\n'` newline char marking the end of each line. Standard out is relatively simple. It is a single area of text shared by all the code in a program. Each printed line is appended at its end.

Standard Out in the Terminal

When you run a program from the terminal, standard out appears right there. Here is example of running the above `hello.py` in the terminal, and whatever it prints appears immediately in the terminal.

```
$ python3 hello.py
hello there
how are you?
I am fine
$
```

Print Function

The Python print() function takes in any number of parameters, and prints them out on one line of text. The items are each converted to text form, separated by spaces, and there is a single `'\n'` at the end (the "newline" char). When called with zero parameters, print() just prints the `'\n'` and nothing else. In the interpreter, standard-out displays to the screen in between the '>>>' prompts, so it's an easy way to see what print() does.

```
>>> print(12, 24, -2)
12 24 -2
>>> print('hi', 'there', -2)
hi there -2
>>> print('woot')   # 1 item, 1 \n
```

```
woot
>>> print()         # 0 items, 1 \n

>>>
```

In Python source code, the text of a string is written with quotes, like `'Hello'`, so we are very accustomed to seeing that form. Note that printing a string, just prints out the **text data** of the string with no quotes:

```
>>> print('Hello')
Hello
>>>
```

Print Option with multiple inputs

You can print out multiple things by calling print with multiple "inputs" each separated by a comma. For example:

```
>>> print('hello', 24)
hello    24
```

Print Option sep=

You are now venturing into "advanced" teritory. These are things you don't need to know! If you are curious, what you can do with print continues.

By default, print() separates the items by tabs. The optional `sep=` parameter sets a different separator text.

```
>>> print(12, 24, -2, sep=':')
12:24:-2
>>> print('but', 'not', 'including', sep='**')
but**not**including
>>> print('but', 'not', 'including', sep='')  # empty string
butnotincluding
```

Print Option end=

By default, print() puts a single `'\n'` after all the items. The optional `end=` parameter sets a custom string to appear after all the items. The most common use of this is when printing a string that already has a `'\n'` at its end. In that case, printing the string ends up double-spacing the output. See how the printed lines each have an extra blank line after them in this example:

```
>>> print('hello\n')
hello
```

```
>>> print('there\n')
there

>>>
```

The issue is that the string has a newline at the end, and then print() adds a second newline, so we get double spacing. The solution is to use `end=` to specify the empty string as the end of each line.

```
>>> print('hello\n', end='')
hello
>>> print('there\n', end='')
there
```

`input()` and Standard In

A program that runs in the terminal is called a "console program" (console and terminal are practically synonyms). These console programs can become much more interactive if we learn a way for the user to give us some "input". Python has a simple function for doing so, the aptly named `input` function.

```
def hello():
    print('hello there')
    user_status = input('how are you? ')
    print('you said: ' + user_status)

def main():
    hello()
```

Input in the Terminal

Here is example of running the above `hello.py` in the terminal. In this example the text written by the user is in **blue**.

```
$ python3 hello.py
hello there
how are you? I am fine
you said: I am fine
```

Changing variable types

When you call the `input` function, the response entered by the user is given back to you! It is always "returned" as a value of type "str" also known as a

string. That is a problem if you want to do any further computation. For example, lets say you wanted to print the value a user entered divided by two:

```
>>> x = input('enter a value: ')
enter a value: 42
>>> print(x)
42
>>> print(x/2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

This is subtle, but recall that every variable has a type. Even though x looks like a number, its type is string. Recall that input always returns a string. The interpreter is complaining that it doesn't make sense to divide a string! Now that you know what the issue is, the fix is simple. You just need to change the variable type to be a float or an int. To do so you can use functions called `float` and `int`.

```
>>> x_float = float(x)
>>> print(x_float/2)
21.0
```

We can call x_float anything we like. I put the term "float" in the name simply so that you can understand the type just by reading the name.

If and Comparisons

This page describes the if-statement, `==` comparisons, if-else and other related structures.

If Statement

The if-statement controls if some lines run or not.

The if-statement syntax has 4 parts (similar to "while"): if, boolean-test, colon, indented body lines:

```
if boolean-test:
  indented body lines
```

If Operation: Evaluate the boolean test, and if it is True, run the "body" lines once. Otherwise skip the body lines, and the run continues after the last body line.

The simplest and most common sort of boolean test uses `==` (two equal signs next to each other) to compare two values, and it's True if the two are the same.

Here is an example that shows an if-statement. The test `num == 6` is evaluated, yielding the boolean value True.

```
>>> num = 6
>>> if num == 6:
...     print('Yay 6!')
Yay 6!
```

If test = vs. == Syntax Error

Note that it's very easy to accidentally type a single equal sign for a comparison like the following, but in Python that is flagged as a syntax error

```
if num = 6:       # typo, meant ==
    print('hi')
```

Aside: in the language C, the above typo runs silently, interpreting the test as always True regardless of the value of i. Finding that tiny typo that can take many, many hours of debugging. The Python behavior is much better: when the `==` is not present, the line is flagged as an error.

Style: Do Not Write x == True

Suppose some foo() function is supposed to return True or False. Do not write an if-test like this:

```
if foo() == True:   # NO NO NO == True
    print('yay')
```

Instead, let the if/while take the boolean value directly like this:

```
if foo():           # YES this way
    print('yay')


# Or to test if foo() is False use not:
if not foo():
    print('not yay')
```

If Else

The optional `else:` part of an if-statement adds code to run in the case that the test is False. Use `else` if the run should do exactly one of action-1 or action-2 dependin gon the test (use regular if to do action-1 or nothing).

```
# set message according to score
if score > high_score:
    message = 'New high score!'
else:
    message = 'Oh well!
```

The else clause is always about picking 1 of 2 actions. To run code if a test is False and otherwise do nothing, just use `not` like this:

```
if not foo():
    message = 'no foo today'
```

Boolean Comparison Operators

The most common way to get a boolean True/False is comparing two values, e.g. the comparison expression `num == 6` evaluates to True when num is 6 and False otherwise.

Comparison operators:

`==` test if two values are equal (2 equals signs together). Works for int, string, list, dict, .. most types. Not recommended to use with float values.

`!=` not-equal, the opposite of equal (uses == under the hood)

`< <=` less-than, less-or-equal. These work for most types that have an ordering: numbers, strings, dates. For strings, `<` provides alphabetic ordering. Uppercase letters are ordered before lowercase. It is generally an error to compare different types for less-than, e.g. this is an error: `4 < 'hello'`

`> >=` greater than, greater-or-equal.

The interpreter example below shows various `==` style comparisons, and for each what boolean value results:

```
>>> 5 == 6
False
>>> 5 == 5
True
>>> 5 != 6
```

```
True
>>> 4 > 2
True
>> 4 > 5
False
>>> 4 > 4
False
>>> 4 >= 4
True
>>> s = 'he' + 'llo'
>>> s == 'hello'
True
>>> 'apple' < 'banana'
True
>>> 'apple' < 4
TypeError: '<' not supported between instances of 'str' and 'int'
```

If Elif

There is a more rarely used **elif** for where a series of if-tests can be strung together (mnemonic: 'elif' is length 4, like 'else'):

```
if s == 'a':
    # a case
elif s == 'b':
    # b case
else:
    # catch-all case
```

The tests are run top to bottom, running the code for the first that is True. However, the logic of when each case runs can be hard to see. What must be true for case c below? You really have to think about the code work work out when (c) happens.

```
if score > high and s != 'alice':
    # a
elif s == 'bob':
    # b
else:
    # c
```

Answer: c happens when s is not 'bob' but also (score <= high or s == 'alice or both)

If/else chains are fine, just don't think they are trivial. Only add **else** if the code needs it. Nice, simple if handles most problems and is the most readable.

Python if - de-facto True/False

The Python if statement looks like this

```
if test:
    print('test is true!')
```

Surprisingly, **any** value can be used in the test expression - string, int, float, list, ...

Python has a de-facto True/False system, where all the "empty" values count as de-facto False: '', None, 0, 0.0, empty-list, empty-dict

Any other value de-facto counts as True: a non-empty string, a non-zero int, a non-empty list. Many languages use this anything-empty-is-false system. The `bool()` function takes any value and returns a formal bool False/True value, so that's a way to see the empty=False interpretation:

```
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool('')   # empty string - False
False
>>> bool([])   # empty list - False
False
>>> bool(None)
False
>>> bool(6)       # non-zero int - True
True
>>> bool('hi')    # non-empty string - True
True
>>> bool([1, 2])  # list of something - True
True
```

Why does the de-facto system exist? It makes it easy to test, for example, for an empty string like the following. Testing for "empty" data is such a common case, it's nice to have a shorthand for it.

```
# long form screen out empty string
if len(s) > 0:
    print(s)


# shorter way, handy!
if s:
    print(s)
```

Boolean Values

The Boolean values in Python are `True` and `False`, typically used to control if-statements and while-loops.

Boolean And Or Not

The Python "bool" type (short for "boolean") has only two values - `True` and `False`. Expressions yielding a bool value like `a > 10`, can be combined with **and or not**, like following (printing 'yay' if a is in the range 0..100 inclusive):

```
if a >= 0 and a <= 100:
    print('yay')
```

Python is unique in using the plain words like "and" for this. many languages use "&&" for and, "||" for or.

a `and` *b* → True if both are True

a `or` *b* → True if one or the other or both are True

`not` *a* → Inverts True/False

Boolean Precedence

There is "precedence" between and/or/not, analogous to arithmetic precedence, where "*" has higher precedence than "+".

Precedence order: `not` is highest precedence, followed by `and`, followed by `or`. Mnemonic: `not` is like unary minus e.g. '-6', `and` is like * mutliplication, `or` is like + addition.

Q: what does the following do:

```
if a < 6 and b < 6 or c < 6:
```

The `and` goes first (higher precedence), so the above is equivalent to the following form with parenthesis added to show the order the comparisons are evaluated:

```
if (a < 6 and b < 6) or c < 6:
```

To force the `or` to go first, put in parenthesis like this:

```
if a < 6 and (b < 6  or c < 6):
```

If you are unsure, you can always add parenthesis to an expression to force the order you want.

Boolean Short-Circuit

Suppose you have an int i, and you want to check if the char at that index is alphabetic .. but only if i is valid. You can write that this way...

```
if i < len(s) and s[i].isalpha():...
```

This works because the boolean evaluation goes left to right, stopping ("short-circuiting") as soon as it can. So in the example above, if `i < len(s)` is False (i.e. i is large), the whole expression evaluates to False. In particular the `s[i].isalpha()` is **not evaluated**. Attempting to evaluate it would be an error, since i is too large, creating an IndexError.

Writing the `i<len(s)` test to the left of the `s[i]` in effect protects it, and this is a common programming pattern. Most computer languages use short circuiting in the their boolean expressions like this.

While Loops

A loop takes a few lines of code, and runs them again and again. Most algorithms have a lines of code that need to be run thousands or millions of times, and loops are the way to do this.

The while-loop uses a boolean test expression to control the run of the body lines. The for-loop is great of looping over a collection. The while-loop is more general, providing enough control for any sort of looping, without requiring a collection to loop over.

While Loop Syntax

The while-loop syntax has 4 parts: while, boolean test expression, colon, indented body lines:

```
while test:
    indented body lines
```

While Operation: Check the boolean test expression, if it is True, run all the "body" lines inside the loop from top to bottom. Then loop back to the top, check the test again, and so on. When the test is False, exit the loop, running continues on the first line after the body lines.

Here is a while loop to print the numbers 0, 1, 2, ... 9 (there are easier ways to do this, but here we're just trying to show the parts of the loop).

```
i = 0
while i < 10:
    print(i)
    i = i + 1
print('All done')
```

```
0
1
2
3
4
5
6
7
8
9
All done
```

Very often the last line of the while body has an "increment" role, such as the `i = i + 1` line above. The test at the top of the loop checks that variable. It's important that on every iteration, the loop advances that variable one step towards the ultimate end of the loop.

While Zero Iterations OK

Just as with the for-loop, a while-loop can iterate zero times. That happens if the boolean test is False the very fist time it is checked, like this:

```
i = 99
while i < 10:
    print(i)
    i += 1
```

```
print('All done')

# (zero iterations - no numbers print at all)
All done
```

Infinite Loop Bug

With a while-loop, it's possible to accidentally write a loop that never exits. In that case, the while just loops and loops but never makes the test False to exit. As the loop runs and runs, the fans on your laptop may spin up as CPU heats up with this high number of lines running without pause.

Here's an infinite loop example caused by a typical looking bug — the variable `i` accidentally stays at the value 1 and the loop just goes forever.

```
i = 0
while i < 10:   # BUG infinite loop
    print(i)
    i = i * 1
print('All done')
```

Don't Forget the Last "Increment" Line

Another easy infinite loop bug is forgetting the `i = i + 1` line entirely, so the variable never advances and the loop never exits. Since the more commonly used for-loop automates the increment step for us, we don't quite have the muscle memory to remember it when writing a while-loop.

Do Not Write == True

Suppose there is some function `foo()` and you want a while loop to run so long as it returns True. Do not write this

```
while foo() == True:   # NO not this way
    ...
```

It's better style to write it the following way, letting the while itself evaluate the True/False of the test:

```
while foo():           # YES this way
    ...
```

Loop Break

The `break` directive in a loop exits the loop immediately. Loops have their standard way of exiting. The `break` gives an extra option to exit the loop if some

special condition occurs. Usually the break is put inside an if that checks for some condition.

This example loops over a list of numbers, printing each one in the usual way. However, the if/break structure checks each number after printing, and breaks out of the loop if the number is 6.

```
nums = [12, 1, 6, 13, 6, 0]
for num in nums:
    if num == 6:
        break  # exit loop immediately
    print(num)
print('All done')
```

```
12
1
All done
```

Most loops do not use break. Break is an option for certain cases where the programmer wants to be able to exit the loop earlier than it would normally.

Loop Continue

The `continue` directive directs the loop run to go back to the top of the loop immediately to start the next iteration. In effect, it skips the current iteration. The continue directive is very rarely used. We mention it here for completeness.

Here is the above example changed to use continue. In effect it skips over iterations where num is 6.

```
nums = [12, 1, 6, 13, 6, 0]
for num in nums:
    if num == 6:
        continue  # jump to top of loop
    print(num)
print('All done')
```

```
12
1
13
0
All done
```

For Loops

A loop takes a few lines of code, and runs them again and again. Most algorithms have a lines of code that need to be run thousands or millions of times, and loops are the way to do this.

For Loop - aka Foreach

The "for" loop is probably the single most useful type of loop. The for-loop, aka "foreach" loop, looks at each element in a collection once. The collection can be any type of collection-like data structure, but the examples below use a list.

Here is a for-loop example that prints a few numbers:

```
>>> for num in [2, 4, 6, 8]:
        print(num)
2
4
6
8
```

Loop Syntax: the loop begins with the keyword `for` followed by a variable name to use in the loop, e.g. `num` in this example. Then the keyword `in` and a collection of elements for the loop, e.g. the list `[2, 4, 6, 8]`. Finally there is colon `:` followed by the indented "body" lines controlled by the loop.

Loop Operation: the loop runs the body lines again and again, once for each element in the collection. Each run of the body is called an "iteration" of the loop. For the first iteration, the variable is set to the first element, and the body lines run (in this case, essentially `num = 2`. For the second iteration, `num = 4` and so on, once for each element.

The main story of the for loop is that if we have a collection of numbers or strings or pixels, the for-loop is an easy way to write code that looks at each value once. Now we'll look at a few features and slightly subtle features of the loop.

Loops with `range()` Function

The python `range()` function creates a collection of numbers on the fly, like [0, 1, 2, 3, 4] This is very useful, since the numbers can be used to index into

collections such as string. The range() function can be called in a few different ways.

range(n) - 1 Parameter Form

The most common form is `range(n)`, for integer n, which returns a numeric series starting with 0 and extending up to but not including n, e.g. `range(6)` returns `0, 1, 2, 3, 4, 5`.

```
>>> for i in range(6):
...   print(i)
...
0
1
2
3
4
5
```

Loop Controls The Variable, Not You

Usually variables only change when we see an assignment with an equal sign `=`

The for-loop is strange, since for each iteration, the loop behind the scenes is setting the variable to point to the next value. Mostly this is very convenient, but it does mean that setting the variable to something at the end of the loop has basically no effect...

```
for num in [2, 4, 6, 8]:
    print(num)
    num = 100    # No effect on output,
                 # the loop resets num on each iteration
```