Python Notes:

Ch 1: Getting Started

Python is case-sensitive

Ch 2: Types, Variables, Simple I/O

**Strings:**

| **Table 2.1. Selected Escape Sequences** | |
| --- | --- |
| **Sequence** | **Description** |
| \\ | Backslash. Prints one backslash. |
| \‘ | Single quote. Prints a single quote. |
| \“ | Double quote. Prints a double quote. |
| \a | Bell. Sounds the system bell. |
| \n | Newline. Moves cursor to beginning of next line. |
| \t | Horizontal tab. Moves cursor forward one tab stop. |

“”” = ASCII exact representation

(“sample”, end=””) = defines string that ends line (\n by default)

+ = concatenation

\* = repetition

\ = line continuation

**Numbers:**

| **Table 2.2. Useful Mathematical Operators** | | | |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Evaluates To** |
| + | Addition | 7 + 3 | 10 |
| - | Subtraction | 7 - 3 | 4 |
| \* | Multiplication | 7 \* 3 | 21 |
| / | Division (True) | 7 / 3 | 2.3333333333333335 |
| // | Division (Integer) | 7 // 3 | 2 |
| % | Modulus | 7 % 3 | 1 |

The decimal module provides support for accurate decimal floating-point arithmetic. To learn more, see the Python documentation.

**Variables:**

 A variable name can contain only numbers, letters, and underscores.

 A variable name can’t start with a number.

Names that begin with an underscore have special meaning in Python.

Try to keep variables under 15 char’s

**String Methods:**

You kick off a method, or invoke it, by adding a dot, followed by the name of the method, followed by a pair of parentheses, after a string value.

| **Table 2.3. Useful String Methods** | |
| --- | --- |
| **Method** | **Description** |
| upper() | Returns the uppercase version of the string. |
| lower() | Returns the lowercase version of the string. |
| swapcase() | Returns a new string where the case of each letter is switched. Uppercase becomes lowercase and lowercase becomes uppercase. |
| capitalize() | Returns a new string where the first letter is capitalized and the rest are lowercase. |
| title() | Returns a new string where the first letter of each word is capitalized and all others are lowercase. |
| strip() | Returns a string where all the white space (tabs, spaces, and newlines) at the beginning and end is removed. |
| replace(old, new [,max]) | Returns a string where occurrences of the string old are replaced with the string new. The optional max limits the number of replacements. |

**Using Different Types:**

When a program produces unintended results but doesn’t crash, it has a **logical error**.

Using the same operator for values of different types is called operator overloading.

Nesting function calls means putting one inside the other. This is perfectly fine as long as the return values of the inner function can be used as argument values by the outer function.

| **Table 2.4. Selected Type Conversion Functions** | | | |
| --- | --- | --- | --- |
| **Function** | **Description** | **Example** | **Returns** |
| float(x) | Returns a floating-point value by converting x | float(“10.0”) | 10.0 |
| int(x) | Returns an integer value by converting x | int(“10”) | 10 |
| str(x) | Returns a string value by converting x | str(10) | ‘10’ |

| **Table 2.5. Useful Augment Assignment Operators** | | |
| --- | --- | --- |
| **Operator** | **Example** | **Is Equivalent To** |
| \*= | x \*= 5 | x = x \* 5 |
| /= | x /= 5 | x = x / 5 |
| %= | x %= 5 | x = x % 5 |
| += | x += 5 | x = x + 5 |
| -= | x -= 5 | x = x - 5 |

Ch 3: Branching, Looping, and Program Planning

**The “if” Statement:**

Syntax: **if** (condition):

(tab) *block*

| Table 3.1. Comparison Operators | | | |
| --- | --- | --- | --- |
| **Operator** | **Meaning** | **Sample Condition** | **Evaluates To** |
| == | equal to | 5 == 5 | True |
| != | not equal to | 8 != 5 | True |
| > | greater than | 3 > 10 | False |
| < | less than | 5 < 8 | True |
| >= | greater than or equal to | 5 >= 10 | False |
| <= | less than or equal to | 5 <= 5 | True |

**The “else” Statement:**

Syntax: **else**:

(tab) *block*

Must have same indentation as if statement above

**The “elif” Statement:**

| Table 3.2. Branching Statements Summary | | |
| --- | --- | --- |
| **Statement** | | **Description** |
| if | <condition>: | if statement. If <condition> is true, <block> is executed; otherwise it’s skipped. |
|  | <block> |  |
| if | <condition>: | if statement with else clause. If <condition> is true, <block1> is executed; otherwise <block2> is executed. |
|  | <block1> |
| else: | |  |
|  | <block 2> |  |
| if | <condition 1>: | if statement with elif clauses and optional final else clause. The block of the first true condition is executed. If no condition is true, the optional else clause’s block is executed. |
|  | <block 1> |
| elif <condition 2>: | |
|  | <block 2> |
| . |  |  |
| . |  |  |
| . |  |  |
| elif <condition N>: | |  |
|  | <block N> |  |
| else: |  |  |
|  | <block N+1> |  |

**The “while” Loop:**

sentry variable: a variable used in the condition and compared to some other value or values

It’s important to initialize your sentry variable.

**Avoiding an Infinite Loop:**

Tracing means you simulate the running of your program and do exactly what it would do, following every statement and keeping track of the values assigned to variables.

**Intentionally Creating an Infinite Loop:**

The break statement means “break out of the loop”

The continue statement means “jump back to the top of the loop.”

You don’t actually need break and continue. Any loop you can write using them can be written without them.

**Logical Operators:**

*and, or,* and *not* are all plain English in Python

Ch 4: For Loops, Strings, and Tuples

ALLCAPS=constant variable

^Simply convention, does not affect variable definitions

None is Python’s way of representing nothing. None makes a good placeholder for a value. It also evaluates to False when treated as a condition.

 the slice word[:4] is exactly the same as word[0:4]

word[2:] is just shorthand for word[2:#] where # is size of string

word[:] is shorthand for word[0:#]

empty tuple example: inventory = ()

Make your programs easier to read by creating tuples across multiple lines. You don’t have to write exactly one element per line, though. It might make sense to write several on a line. Just end each line at one of the commas separating elements and you’ll be fine.

Other programming languages offer structures similar to tuples. Some go by the name “arrays” or “vectors.” However, those other languages usually restrict the elements of these sequences to just one type. So, for example, you couldn’t mix strings and numbers together. Just be aware that these other structures don’t usually offer all the flexibility that Python sequences do.

Ch 5: Lists and Dictionaries

You can assign an existing list element a new value with indexing, but you can’t create a new element in this way. An attempt to assign a value to a nonexistent element will result in an error.

Watch out when you use the remove() method. If you try to remove a value that isn’t in a list, you’ll generate an error.

| Table 5.1. Selected List Methods | |
| --- | --- |
| **Method** | **Description** |
| append(***value***) | Adds ***value*** to end of a list. |
| sort() | Sorts the elements, smallest value first. Optionally, you can pass a Boolean value to the parameter reverse. If you passTrue, the list will be sorted with the largest value first. |
| reverse() | Reverses the order of a list. |
| count(***value***) | Returns the number of occurrences of***value***. |
| index(***value***) | Returns the first position number of where ***value*** occurs. |
| insert(***i, value***) | Inserts ***value*** at position ***i***. |
| pop([***i***]) | Returns value at position ***i*** and removes value from the list. Providing the position number ***i*** is optional. Without it, the last element in the list is removed and returned. |
| remove(***value***) | Removes the first occurrence of ***value***from the list. |

* Tuples are faster than lists. Because the computer knows they won’t change, tuples can be stored in a way that makes using them faster than using lists. For simple programs, this speed difference won’t matter, but in more complex applications, with very large sequences of information, it could.
* Tuples’ immutability makes them perfect for creating constants since they can’t change. Using tuples can add a level of safety and clarity to your code.
* Sometimes tuples are required. In some cases, Python requires immutable values. Okay, you haven’t actually seen any of those cases yet, but there is a common situation you’ll see when you learn about dictionaries, later in this chapter in the “Using Dictionaries” section. Dictionaries require immutable types, so tuples will be essential when creating some kinds of dictionaries.

But, because lists are so flexible, you’re probably best off using them rather than tuples the majority of the time.

#### Unpacking a Sequence

If you know how many elements are in a sequence, you can assign each to its own variable in a single line of code:

>>> name, score = ("Shemp", 175)

>>> print(name)

Shemp

>>> print(score)

175

This is called *unpacking* and works with any sequence type. Just remember to use the same number of variables as elements in the sequence, because otherwise you’ll generate an error.

So, the moral of this story is: be aware of shared references when using mutable values. If you change the value through one variable, it will be changed for all.

However, you can avoid this effect if you make a copy of a list through slicing.

One thing that sometimes trips up beginning programmers is that a value can’t be used to get a key in a dictionary.

There’s another way to retrieve a value from a dictionary. You can use the dictionary method get(). The method has a built-in safety net for handling situations where you ask for a value of a key that doesn’t exist. If the key doesn’t exist, the method returns a default value, which you can define. Take a look at another attempt:

>>> print(geek.get("Dancing Baloney", "I have no idea."))

I have no idea.

There are a few things you should keep in mind when creating dictionaries:

* A dictionary can’t contain multiple items with the same key. Think again about a real dictionary. It becomes pretty meaningless if you can keep adding the same word with totally new definitions whenever you want.
* A key has to be immutable. It can be a string, a number, or a tuple, which gives you lots of possibilities. A key has to be immutable because, if it weren’t, you could sneak into a dictionary later and change its keys, possibly ending up with two identical keys. And you just learned you can’t have that!
* Values don’t have to be unique. Also, values can be mutable or immutable. They can be anything you want.

There’s even more you can do with dictionaries. [**Table 5.2**](javascript:moveTo('ch05tab02');) summarizes some useful methods that can help you get more out of this type.

| **Table 5.2. Selected Dictionary Methods** | |
| --- | --- |
| **Method** | **Description** |
| get(***key***, [***default***]) | Returns the value of ***key***. If ***key*** doesn’t exist, then the optional ***default*** is returned. If ***key*** doesn’t exist and***default*** isn’t specified, then None is returned. |
| keys() | Returns a view of all the keys in a dictionary. |
| values() | Returns a view of all the values in a dictionary. |
| items() | Returns a view of all the items in a dictionary. Each item is a two-element tuple, where the first element is a key and the second element is the key’s value. |

Ch 6: Functions

I began the definition of my new function with a single line:

def instructions():

A function always ends after it hits a return statement.

|  |  |
| --- | --- |
|  | Make sure to have enough variables to catch all the return values of a function. If you don’t have the right number when you try to assign them, you’ll generate an error. |

If you just list out a series of variable names in a function’s header, you create positional parameters:

def birthday1(name, age):

If you call a function with just a series of values, you create positional arguments:

birthday1("Jackson", 1)

You can combine keyword arguments and positional arguments in a single function call, but this can get tricky. Once you use a keyword argument, all the remaining arguments in the call must be keyword arguments, too. To keep things simple, try to use all keyword or all positional arguments in your function calls.

Once you assign a default value to a parameter in the list, you have to assign default values to all the parameters listed after it.

Global constants (global variables that you treat as constants), on the other hand, can make programs less confusing. For example, say you’re writing a business application that calculates someone’s taxes. Like a good programmer, you have written a variety of functions in your code, all of which use the somewhat cryptic value .28as the tax rate. Instead, you could create a global constant called TAX\_RATE and set it to .28. Then, in each function, you could replace the number .28 with TAX\_RATE. This produces two benefits. It makes your code clearer and it makes changes (like a new tax rate) no sweat.

Any time you get a mutable value passed to a function, you have to be careful. If you know you’re going to change the value as you work with it, make a copy and use that instead.

Ch 7: Files and Exceptions

| Table 7.1. Selected Text File Access Modes | |
| --- | --- |
| **Mode** | **Description** |
| “r” | Read from a text file. If the file doesn’t exist, Python will complain with an error. |
| “w” | Write to a text file. If the file exists, its contents are overwritten. If the file doesn’t exist, it’s created. |
| “a” | Append a text file. If the file exists, new data is appended to it. If the file doesn’t exist, it’s created. |
| “r+” | Read from and write to a text file. If the file doesn’t exist, Python will complain with an error. |
| “w+” | Write to and read from a text file. If the file exists, its contents are overwritten. If the file doesn’t exist, it’s created. |
| “a+” | Append and read from a text file. If the file exists, new data is appended to it. If the file doesn’t exist, it’s created. |

Python remembers where I last left off. It’s like the computer puts a bookmark in the file and each subsequent read() begins where the last ended. When you read to the end of a file, subsequent reads return the empty string.

To start back at the beginning of a file, you can close and open it.

The readline() method lets you read characters from the current line. You just pass the number of characters you want read from the current line and the method returns them as a string. If you don’t pass a number, the method reads from the current position to the end of the line. Once you read all of the characters of a line, the next line becomes the current line.

| **Table 7.2. Selected File Object Methods** | |
| --- | --- |
| **Method** | **Description** |
| close() | Closes the file. A closed file cannot be read from or written to until opened again. |
| read([***size***]) | Reads ***size*** characters from a file and returns them as a string. If size is not specified, the method returns all of the characters from the current position to the end of the file. |
| readline([***size***]) | Reads ***size*** characters from the current line in a file and returns them as a string. If size is not specified, the method returns all of the characters from the current position to the end of the line. |
| readlines() | Reads all of the lines in a file and returns them as elements in a list. |
| write(***output***) | Writes the string ***output*** to a file. |
| writelines(***output***) | Writes the strings in the list ***output*** to a file. |

| **Table 7.3. Selected Binary File Access Modes** | |
| --- | --- |
| **Mode** | **Description** |
| “rb” | Read from a binary file. If the file doesn’t exist, Python will complain with an error. |
| “wb” | Write to a binary file. If the file exists, its contents are overwritten. If the file doesn’t exist, it’s created. |
| “ab” | Append a binary file. If the file exists, new data is appended to it. If the file doesn’t exist, it’s created. |
| “rb+” | Read from and write to a binary file. If the file doesn’t exist, Python will complain with an error. |
| “wb+” | Write to and read from a binary file. If the file exists, its contents are overwritten. If the file doesn’t exist, it’s created. |
| “ab+” | Append and read from a binary file. If the file exists, new data is appended to it. If the file doesn’t exist, it’s created. |

| **Table 7.4. Selected Pickle Functions** | |
| --- | --- |
| **Function** | **Description** |
| dump(object, file, [,bin]) | Writes pickled version of object to file. If bin isTrue, object is written in binary format. If bin isFalse, object is written in less efficient, but more human- readable, text format. The default value ofbin is equal to False. |
| load(file) | Unpickles and returns the next pickled object infile. |

| **Table 7.5. Shelve Access Modes** | |
| --- | --- |
| **Mode** | **Description** |
| “c” | Open a file for reading or writing. If the file doesn’t exist, it’s created. |
| “n” | Create a new file for reading or writing. If the file exists, its contents are overwritten. |
| “r” | Read from a file. If the file doesn’t exist, Python will complain with an error. |
| “w” | Write to a file. If the file doesn’t exist, Python will complain with an error. |

|  |  |
| --- | --- |
|  | While you could simulate a shelf by pickling a dictionary, the shelve module is more memory efficient. So, if you need random access to pickled objects, create a shelf. |
| **Table 7.6. Selected Exception Types** | |
| **Exception Type** | **Description** |
| IOError | Raised when an I/O operation fails, such as when an attempt is made to open a nonexistent file in read mode. |
| IndexError | Raised when a sequence is indexed with a number of a nonexistent element. |
| KeyError | Raised when a dictionary key is not found. |
| NameError | Raised when a name (of a variable or function, for example) is not found. |
| SyntaxError | Raised when a syntax error is encountered. |
| TypeError | Raised when a built-in operation or function is applied to an object of inappropriate type. |
| ValueError | Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value. |
| ZeroDivisionError | Raised when the second argument of a division or modulo operation is zero. |

A single piece of code can result in different types of exceptions. Fortunately, you can trap for multiple exception types. One way to trap for multiple exception types is to list them in a single except clause as a comma-separated group enclosed in a set of parentheses.

Ch 8: Software Objects

Objects are created (or *instantiated* in OOP-speak) from a definition called a *class*—programming code that can define attributes and methods. Classes are like blueprints. A class isn’t an object, it’s a design for one. And just as a foreman can create many houses from the same blueprint, a programmer can create many objects from the same class. As a result, each object (also called an *instance*) instantiated from the same class will have a similar structure.

You’ll notice that my class name begins with a capital letter. Python doesn’t require this, but it’s the standard convention, so you should begin all your class names with a capital letter.

Every *instance method*—a method that every object of a class has—must have a special first parameter, called self by convention.

There’s a special method you can write, called a *constructor*, that is automatically invoked right after a new object is created.  As a constructor method, \_\_init\_\_() is automatically called by any newly created Critter object right after the object springs to life.

Python has a collection of built-in “special methods” whose names begin and end with two underscores, like \_\_init\_\_, the constructor method.