Week 5 - Friday

COMP 1800

Last time

- What did we talk about last time?
- Dictionaries
- Statistics
  - Mean
  - Median
  - Mode

Questions?

Assignment 4

Review

Exam 1

- Format:
  - Multiple choice questions (~20%)
  - Short answer questions (~20%)
  - Programming problems (~60%)
- Written in class
  - No notes
  - Closed book
  - No calculator

Python Basics

Problem solving

- The famous mathematics educator George Pólya outlined a series of steps for solving problems:
  1. Understand the problem
  2. Make a plan
  3. Execute the plan
  4. Look back and reflect

Literals

- Each type has losts of **literals** associated with it
- A **literal** is a concrete value within a given type

Type	Examples	Notes

Integer	`37`
	`0`
	`-15`	

Floating-point values	`2.75` `6.02E23`	Scientific notation is allowed
Strings	`'Trouble'` `"Funk"`	Either single quotes or double quotes can be used
Boolean	`True` `False`	
Lists	`[2, 3, 5, 7, 11, 13]`	Marked with square brackets

Dictionaries	`{1 : 'Fellowship of the Ring',`
	`2 : 'Two Towers',`
	`3 : 'Return of the King'}`
		Marked with curly braces

Variables

- It is also possible to create **variables** for each data type
- Think of a variable as a "box" that you can put values into
- The name of a variable is an **identifier**
- The type of a variable is whatever you've most recently put into it
- The following creates a variable called **`i`** that currently holds an integer
- Then, we multiply **`i`** by 3 and put it in another variable

```
i = 42
tripled = 3 * i
```

Changing the value of a variable

- Python variables are not like variables in math which have a fixed (but unknown) value
- Instead, a Python variable can be changed by a line of code
- We use the **assignment operator (=)** to change the value of a variable as follows:

```
i = 42
print(3 * i) # prints 126
i = 5
print(3 * i) # prints 15
```

- The first time, `3 * i` is 126, but the second time, it's 15

Variable names

- Variables have to start with a letter (or an underscore) and then can have letters, underscores, or numbers

```
rainfall = 2.61
top10 = 21
5fingers = 19
first number = 42
```

Legal

Illegal

- Spaces aren't allowed in a variable name
- The book uses camel-case
  - To make it more readable, each word in a multi-word variable name is capitalized

```
awesomeVariable = 15
```

Operators

- Python has a number of operators that work with integers and floating-point (decimal) numbers

Operator	Operation	Example	Result
+	Add	5 + 7	12
-	Subtract	5 - 7	-2
*	Multiply	5 * 7	35
//	Integer division (round down)	5 // 7	0
/	Regular division	5 / 7	0.7142857142857143
%	Modulo (remainder)	5 % 7	5
**	Exponentiation	5 ** 7	78125

Order of operations

- You can do complicated expressions
- Just like math, there's an order of operations that determines which operations happen first

Operator	Operation	Evaluation Order
`()`	Parentheses	Left to right
`**`	Exponentiation	Right to left
`* // / %`	Multiplication and division	Left to right
`+ -`	Addition and subtraction	Left to right

```
print(3 * (4 + 2) / 8) # prints 2.25
```

Converting between integers and floating-point

- In math, there's no difference between 3 and 3.0
- In Python, the difference is there, but it's subtle
- You can convert between the integers and floating-point variables using the following functions

Function	Description	Example	Result
`float(number)`	Convert to floating-point	`float(15)`	`15.0`
`int(value)`	Convert to integer, dropping fractional part	`int(2.7)`	`2`
`round(value)`	Round to the nearest integer	`round(2.7)`	`3`

Output

- Basic output is done with **`print()`**
- Put what you want to print inside the parentheses
- You can print:
  - Any text enclosed in single or double quotes:

```
print('43 eggplants')
```

  - Any integer:

```
print(43)
```

  - Any floating-point number:

```
print(23.984)
```

  - Even complex numbers:

```
print(5 + 7j)
```

- If you want multiple things to go on the same line, you can use **`print()`** with more than one argument:

```
print(99, 'red', 'balloons')
```

- By default, they will be printed with a space between each one

Case Sensitivity

- Python is a case sensitive language
- **`Print`** is not the same as **`print`**
- **`print('Word!')`** prints correctly
- **`Print('Word!')`** causes an error

Whitespace

- Python doesn't care about whitespace **within** a line of code

```
print('Hello, world!')
```

is the same as:

```
print     (    'Hello, world!'    )
```

- However, whitespace at the **beginning** of a line of code **matters**!
- The following will cause an error:

```
    print('Hello, world!')
```

- Indentation is important in Python, so don't indent without reason!

Comments

- Single line comments use **#**
- Everything after the **#** is a comment and doesn't affect the execution of the program

```
print('Hi!') # this is a comment
```

- Sometimes, you want to comment out a section of code to see what happens if it doesn't run
- To do that in Python, put triple apostrophe (`'''`) on a line by itself before the code and another after

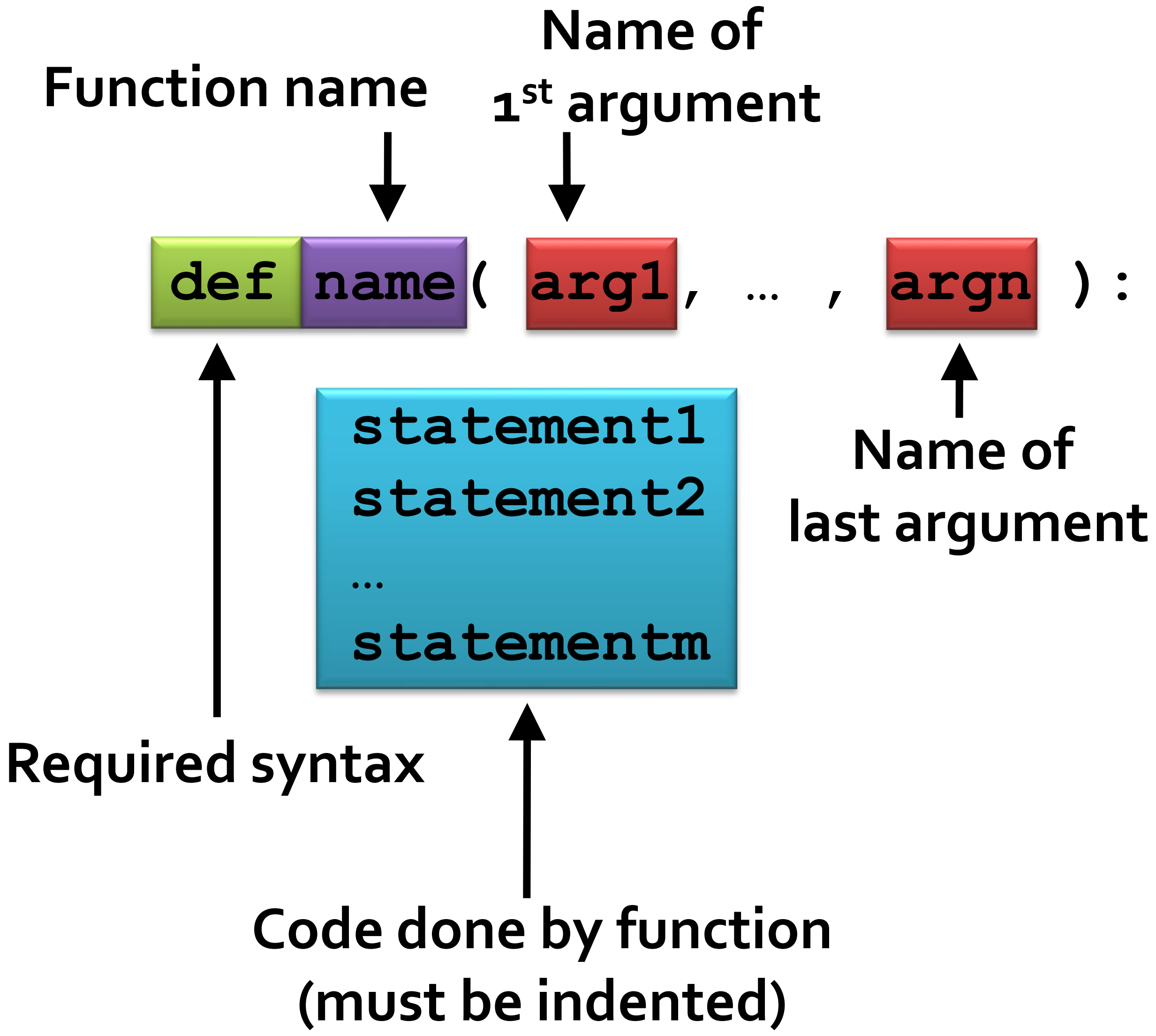
```
'''
print('Hi!')
print('Bye!)
print('No one will see this!')
'''
```

Functions and Libraries

Functions

- A really powerful tool in most programming languages is the ability to package up some code into a chunk that you can use over and over
- This idea has different names in different languages:
  - Function
  - Method
  - Subroutine
  - Procedure
- A key feature of functions is that they can take zero or more arguments that allow you to tell the function to do different things

Defining a function



`return` statements

- Like most code in Python, the code inside of a function executes line by line
- Of course, you are allowed to put loops inside functions
- You can also put in **`return`** statements
- A function will stop executing and jump back to wherever it was called from when it hits a **`return`**
- The **`return`** statement is where you put the value that will be given back to the caller

Turtle

- Turtle is a tool that lets us draw simple pictures in Python
- To use Turtle, we first have to import the turtle library

```
import turtle
```

- Then, we have to create a turtle
- I name mine **`yertle`**, but you can name it any legal variable

```
yertle = turtle.Turtle()
```

- Don't worry too much about this syntax

Methods

- A turtle object has **methods**
- Methods let us tell the turtle to do things or ask it questions
- To call a method, you say the name of the turtle (**`yertle`**, in my case), you put a dot, then you put the name of the method you want, then parentheses, and sometimes information between the parentheses
  - The extra information are called **parameters**
- For example, to make **`yertle`** move forward 100 steps, type:

```
yertle.forward(100)
```

Turtle methods

- The book has a much longer list, but here are a few useful turtle methods

Method	Parameter	Description
`forward`	Distance	Move forward
`backward`	Distance	Move backward
`left`	Angle	Turn counter-clockwise
`right`	Angle	Turn clockwise
`up`	None	Pick up the turtle's tail (to stop drawing)
`down`	None	Put down the turtle's tail (to draw again)
`heading`	None	Return the angle the turtle is pointing
`position`	None	Return the position of the turtle

Using `math` functions

- First, import **`math`** at the top of your program
- After importing **`math`**, you still say **`math.`** before the name of a function
- For example, to compute the cosine of 2.6 radians, you can do the following:

```
import math
result = math.cos(2.6)
```

- Note that all the trigonometry functions take radians, not degrees

Some math functions

Return type	Name	Job
Integer	`ceil( x )`	Find the ceiling of x
Integer	`floor( x )`	Find the floor of x
Floating-point	`fabs( x )`	Find the absolute value of x
Floating-point	`sin( theta )`	Find the sine of angle theta
Floating-point	`cos( theta )`	Find the cosine of angle theta
Floating-point	`tan( theta )`	Find the tangent of angle theta
Floating-point	`exp( a )`	Raise *e* to the power of a (e^a)
Floating-point	`log( a )`	Find the natural log of a
Floating-point	`pow( a, b )`	Raise a to the power of b (a^b)
Floating-point	`sqrt( a )`	Find the square root of a
Floating-point	`degrees( radians )`	Convert radians to degrees
Floating-point	`radians( degrees )`	Convert degrees to radians

Another useful library

- The **`random`** library lets us produce random numbers
- It has two functions that will be useful to us:
  - **`randint(a, b)`** : Returns a random integer **n** where **a** ≤ **n** ≤ **b**
  - **`random()`** : Returns a random floating-point value from [0, 1)
- To use them, import **`random`** and then call the functions qualified by **`random`** followed by a period:

```
import random

dice = random.randint(1, 6)
percentage = random.random()
```

Loops

`for` loops

- Often, we want to repeat something
- The easiest way to do that in Python is with a **`for`** loop:

```
for i in range(n):
  statement1
  statement2
  statement3
  …
```

- All the statements in the **`for`** loop are repeated **`n`** times

The `range()` function

- The **`range()`** function produces a sequence of values that a variable will take on
- With only a single parameter **`n`**, the sequences of numbers is 0, `1`, 2,…,**`n`** – `1` (but not **`n`**)

```
for i in range(100):
```

- With two parameters, **`a`** and **`b`**, the sequence starts at **`a`** and goes up to **`b`** – `1` (but not **`b`**)

```
for i in range(10,20):
```

- With three parameters, **`a`**, **`b`**, and **`step`**, the sequence starts at **`a`** and goes almost up to (but not including) **`b`**, taking steps of size **`step`**

```
for i in range(10,20,5):
```

Patterns

- A **design pattern** is a problem-solving technique in coding in which there is a standard way of do something that is used a lot
- **Accumulator Pattern**
  - Produce a result by iterating over a sequence of values and accumulate their sum (or other aggregation) along the way
- This example finds the sum of numbers from 1 up to 10:

```
acc = 0
for x in range(1, 11):
    acc = acc + x
```

Selection Statements

Behold!

- To make choices in our program, we can use an `if`-statement:

```
x = 4

if x < 5:
  print('x is small!')
```

- `x is small` will only print out if `x` is less than 5
- In this case, we know that it is, but `x` could come from user input or a file or elsewhere

Anatomy of an `if`

keyword `if`

Any Boolean expression

```
if condition :
    statement(s)
```

Any executable statements

Note: The colon after the condition and the indentation before the statement are **required**

Comparison

- The most common condition you will find is a comparison between two things
- In Python, that comparison can be:
  - `==` equals
  - `!=` does not equal
  - `<` less than
  - `<=` less than or equal to
  - `>` greater than
  - `>=` greater than or equal to
- These are called **relational** operators

and and **or**

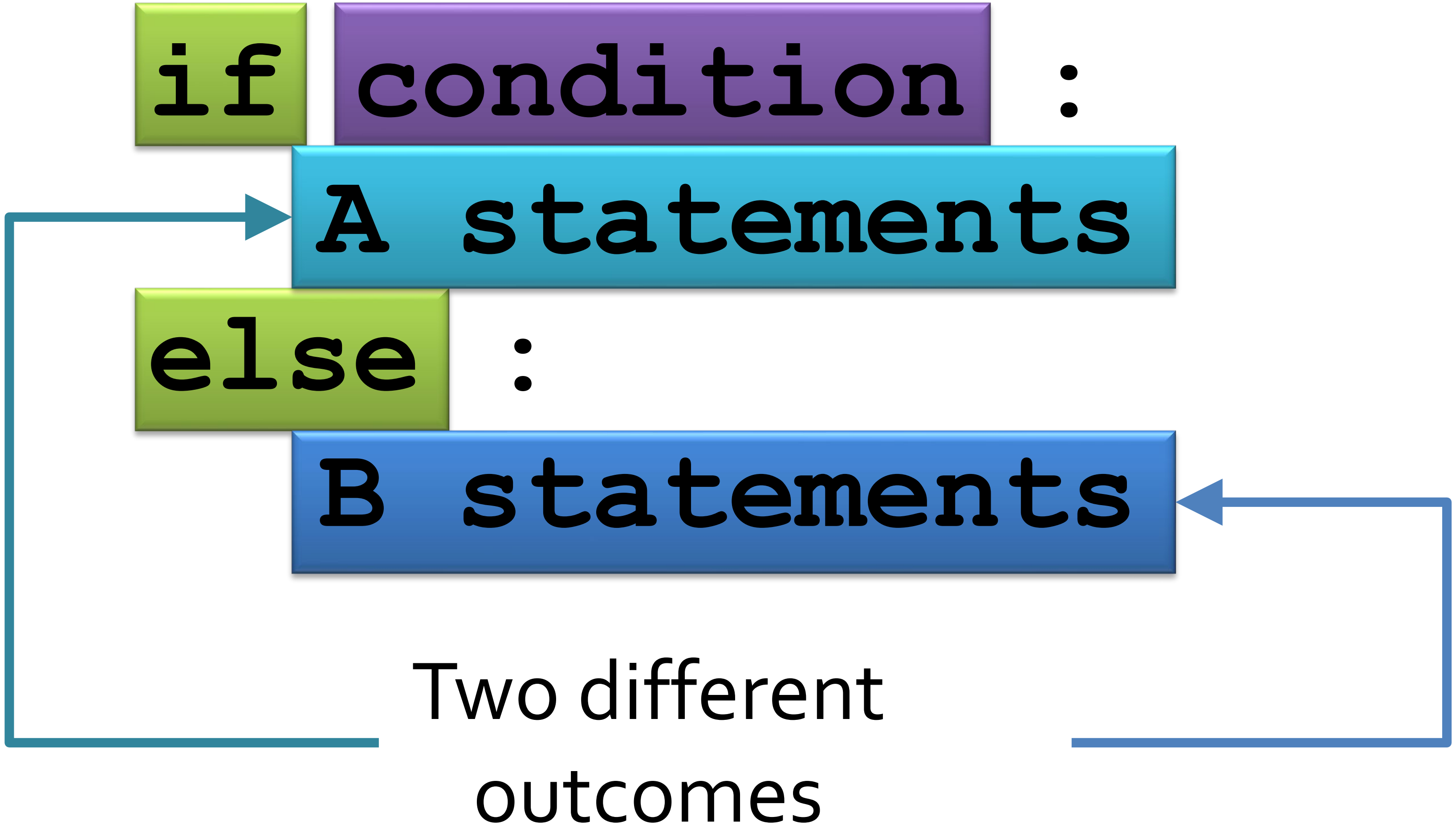
- You can also have multiple Boolean conditions in an **`if`** statement
- You can join them together with:
  - **`and`** (which results in a **`True`** value only if both the conditions it joins are **`True`**)
  - **`or`** (which result in a **`True`** value if either of the conditions it joins are **`True`**)

```
if attempts < 5 and password == 'open sesame':
  print('You know the secret!')
```

Either/Or

- Sometimes you have to make a decision
- If a condition is true, you go one way, if not, you go the other
- Both outcomes cannot happen
- For these situations, we use the `else` construct

Anatomy of an `if-else`



else example

```
if balance < 0:
  print('You are in debt!')
else:
  print('You have $' + str(balance))
```

`if` and `elif`

- What if you have a list of mutually exclusive conditions?
- You can tie all the possibilities together starting with **`if`**, then for each additional condition, you use **`elif`** to check it, and then you can optionally end with an **`else`** if none of the other conditions were met

```
if index == 1:
    print('First')
elif index == 2:
    print('Second')
elif index == 3:
    print('Third')
else:
    print(str(index) + "th")
```

Strings

The string type

- The string type can hold any number of characters, not just a single letter
- A string literal is what we use whenever we print out text
- Strings can store text (up to some pretty large length, billions of characters on 32-bit Python and much more in 64-bit) from *most* of the different scripts in the world

```
text = 'message in a bottle'
```

Single vs. double quotes

- In Python, there's no difference at all between single quotes and double quotes

```
word1 = 'eggplant'
word2 = "eggplant" # exactly the same
```

- If the string contains single quotes, we'll usually use double quotes

```
message = "He earned an 'A'"
```

- Likewise, a string that contains double quotes is usually written with single quotes

```
sentence = 'Bob said, "I refuse."'
```

String operations

- You can use **+** to concatenate two strings together (to get a third string that is both of them stuck together)

```
place = 'boon' + 'docks'
print(place) # prints boondocks
```

- You can use ***** to get repetitions of a string

```
comment = 'yeah ' * 3
print(comment) # prints yeah yeah yeah
```

String operations continued

- You can use the **`len()`** function to get the length of a string

```
author = 'Thomas Pynchon'
print( len(author) ) # prints 14
```

- You can use square brackets to get a particular character in the string
- Indexes start at 0
  - The first character in a string is at 0, the last is at its length - 1

```
movie = 'Dr. Strangelove'
print(movie[4]) # prints S
```

Indexing backwards

- You can also index from the back of a string instead of the front
- To do so, use negative numbers, where -1 is the last character in the string, -2 is the second to last, and so on

```
book = 'Harry Potter'
print(book[-1]) # prints r
print(book[-6]) # prints P
```

- Be careful! If you index past the end or the beginning of the string, your code will have an error

```
word = 'wombat'
print(word[6])  # error
print(word[-8]) # error
```

Slices

- If you want to get a substring (a part of a string) from a string, you can use the **slice** notation
  - Two numbers with a colon (`:`) in between
  - The first number is the starting point, the second number is the location after the ending point
  - If you subtract the first from the last, you'll get the length of the result

```
adjective = 'dysfunctional'
noun = adjective[3:6] # noun contains 'fun'
```

More on slices

- You can leave off the first index and Python will assume 0

```
word = 'things'
width = word[:4] #width contains 'thin'
```

- Or you can leave off the last index and Python will assume the length of the string

```
intelligence = 'smart'
craft = intelligence[2:] #craft contains 'art'
```

The in operator

- The **`in`** operator will give a **`True`** if a string can be found inside another string and **`False`** otherwise
- This can be useful in **`if`** statements

```
animal = 'jellyfish'
if 'fish' in animal:
     print ("It's a fish!")
else:
     print ('No fish here.')
```

- You can also use **`not in`** if you want to see if a string is not inside another strong

Iterating over a string

- We can use a **for** loop to iterate over all the characters in a string by using the length of the string and indexing into it

```
for i in range(len(text)):
  print(text[i])
```

- We can also iterate over all the characters in a string directly with a **for** loop

```
for letter in text: # equivalent to loop above
  print(letter)
```

`ord()` and `chr()`

- We can convert a string with a single character in it into the integer that represents it with the **`ord()`** function

```
number = ord('a') # number contains 97
```

- If you know the numerical value of a character, you can convert that number back into a string using the **`chr()`** function

```
letter = chr(100) # letter contains 'd'
```

ASCII table

- Everything in the computer is 1's and 0's
- Each character has a number associated with it
- These numbers are sometimes listed in tables
- The ASCII table only covers 7 bits of information (0-127)
- **NEVER EVER TYPE THESE NUMBERS IN CODE**
- What's important to know:
  - All the characters are numbered
  - The uppercase letters are contiguous
  - The lowercase letters are contiguous
  - The numerical digits are contiguous

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

Cryptography

Encryption and decryption

- **Encryption** is the process of taking a message and encoding it
- **Decryption** is the process of decoding the code back into a message
- A **plaintext** is a message before encryption
- A **ciphertext** is the message in encrypted form
- A **key** is an extra piece of information used in the encryption process

Transposition cipher: Rail Fence Cipher

- In the rail fence cipher, a message is written vertically along a fixed number of "rails," wrapping back to the top when the bottom is reached
- To finish the encryption, the message is stored horizontally
- This is also known as a **columnar transposition**
- Encryption of "WE ARE DISCOVERED, FLEE AT ONCE" with three rails:

W	R	I	O	R	F	E	O	E
E	E	S	V	E	L	A	N	X
A	D	C	E	D	E	T	C	J

- **Ciphertext:** WRIORFEOEEESVELANXADCEDETCJ

Shift cipher

- A shift cipher encrypts a message by shifting all of the letters down in the alphabet
- Using the Latin alphabet, there are 26 (well, 25) possible shift ciphers
- We can model a shift cipher by thinking of the letters A, B, C, … Z as 0, 1, 2, … 25
- Then, we let the key $\boldsymbol{k}$ be the shift
- For a given letter with value $\boldsymbol{x}$:

$$\boldsymbol{encrypt}(\boldsymbol{x}) = (\boldsymbol{x} + \boldsymbol{k}) \bmod 26$$

Example: Caesar Cipher

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

- ***E***("KILL EDWARD") = "NLOO HGZDUG"
- What is ***E***("I DRINK YOUR MILKSHAKE")?
- What is ***D***("EUHDNLWGRZQ")?
- This code was actually used by Julius Caesar who used it to send messages to his generals

Vigenère cipher

- The Vigenère cipher is a form of polyalphabetic substitution cipher
- In this cipher, we take a key word and "add" its letters to our message
- Assuming letter values are in the range 0-25
  - Add them together
  - Mod by 26 to keep them in the range 0-25
  - If they're not in that range, convert them to that range and then back
- If the message is longer than the keyword, we start the keyword over again

Vigenère example

- Key: BENCH
- Plaintext: A LIMERICK PACKS LAUGHS ANATOMICAL

B		E	N	C	H	B	E	N	C		H	B	E	N	C		H	B	E	N	C	H		B	E	N	C	H	B	E	N	C	H
A		L	I	M	E	R	I	C	K		P	A	C	K	S		L	A	U	G	H	S		A	N	A	T	O	M	I	C	A	L
B		**P**	**V**	**O**	**L**	**S**	**M**	**P**	**M**		**W**	**B**	**G**	**X**	**U**		**S**	**B**	**Y**	**T**	**J**	**Z**		**B**	**R**	**N**	**V**	**V**	**N**	**M**	**P**	**C**	**S**

Lists

Lists

- Python provides a way to make lists of general objects
- To make a list, you can put a collection of objects inside square brackets

```
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
'Friday', 'Saturday', 'Sunday']
```

- They can even be different types

```
stuff = ['Danger!', 3, True, 1.7]
```

Lists

- Using the terminology introduced before, lists are:
  - **Heterogeneous**: you can put different kinds of data into a list, but Python programmers usually try not to do this, since it's confusing
  - **Sequential**: items are stored in a particular order
  - **Mutable**: individual items can be changed, and the size of the list can be changed
  - Delimited by **[ ]**
  - Indexed by integer position using **[ ]**
  - Negative indexing is allowed
  - Slicing with **[:]** is supported

Accessing an element

- As with strings, use square brackets and a number to access an element in the list

```
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
'Friday', 'Saturday', 'Sunday']
bleh = days[0] # contains 'Monday'
```

- Like strings, elements are numbered from 0 to the length – 1
- You can use the **`len()`** function to get the length of a list

```
count = len(days)   # contains 7
```

Changing elements in a list

- You can also change the elements in a list using the square bracket notation

```
birds = ['Duck', 'Duck', 'Duck']
birds[2] = 'Goose'
print(birds) #prints ['Duck', 'Duck', 'Goose']
```

- This is one of the bigger differences between strings and general lists
- You **cannot** change the characters in a string
- You have to make a new string

Slices on lists

- Just like strings, you can use the slice notation to get a copy of part of a list

```
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
'Friday', 'Saturday', 'Sunday']
weekend = days[5:7]
print(weekend) #prints ['Saturday', 'Sunday']
```

- The same shortcuts for string slices still work:
  - Python assumes 0 if you leave off the first number
  - It assumes the length if you leave off the last number

```
weekdays = days[:5] #Monday through Friday
```

Multiplying a list

- Like with strings, you can multiply a list by an integer to get a list made up of multiple copies of the list repeated

```
greeting = ['Hello']
manyGreetings = greeting * 5
# manyGreetings contains:
# ['Hello', 'Hello', 'Hello', 'Hello', 'Hello']
```

Empty list

- Just like an empty string, you can have an empty list

```
data = []
```

- Such a list contains no items and has a length of zero

```
length = len(data)
print(length) #prints 0
```

- Why would you want an empty list?

Adding elements to a list

- You can add elements to a list, empty or otherwise
- One way is by using the **`append()`** method, which adds elements to the end of a list

```
data = []
data.append(3)
data.append(7)
data.append(8)
print(data) # prints [3, 7, 8]
```

- There are other ways to add (and remove) items from a list

Useful list methods

Method	Example	Description
`list`	`list(range(100))`	Make a list from the given sequence
`append`	`items.append('goat')`	Add an item to the end of the list
`insert`	`items.insert(4, 'thing')`	Insert an item at a location, moving everything else down
`pop`	`items.pop()`	Remove the last item in the list and return it
`pop`	`items.pop(5)`	Remove item at a given location ad return it
`sort`	`items.sort()`	Sort the list
`reverse`	`items.reverse()`	Reverse the list
`index`	`items.index('walnut')`	Return the first location where an item can be found
`count`	`items.count('apple')`	Count the occurrences of an item
`remove`	`items.remove('goat')`	Remove the first occurrence of an item
`clear`	`items.clear()`	Remove everything from a list

Looping over the contents of lists

- Just as with strings, we can use a **`for`** loop to iterate over everything in a list
- Directly:

```
for item in list:
     print(item)
```

- Or by using an index:

```
for i in range(len(list)):
     print(list[i])
```

- The first version is simpler, but sometimes we need to know the index

Dictionaries

Dictionaries

- A dictionary goes by many names:
  - Map
  - Lookup table
  - Symbol table
- The idea is a table that has a two columns, a key and a value
- You can store, lookup, and change the value based on the key

Dictionaries in Python

- You can create a dictionary in Python
  - Enclosed in curly braces (**{ }**)
  - With a colon (**:**) between each key-value pair

```
superheroes = {'Spiderman' : 'Climbing and webs',
 'Wolverine' : 'Super healing', 'Professor X' :
 'Telepathy', 'Human Torch' :
 'Flames and flying', 'Deadpool' :
 'Super healing', 'Mr. Fantastic' : 'Stretchiness'}
```

Accessing values by key

- Like lists, you can index into a dictionary with square brackets
- **Unlike** lists, you put the key into the square brackets, not a number

```
print(superheroes['Spiderman'])
# prints 'Climbing and webs'
```

- You can also change the value for a given key with square brackets

```
superheroes['Spiderman'] = 'Science stuff'
```

Keys and values

- Dictionaries allow you to get a data structure that contains all the keys using the **keys()** method

```
print(superheroes.keys())
# 'Spiderman', 'Wolverine', etc.
```

- You can also get all the values using the `values()` method

```
print(superheroes.values())
# 'Climbing and webs', 'Super healing', etc.
```

- These structures aren't lists, but you can iterate over them with a `for` loop

```
for key in superheroes.keys():
  print(key)
```

Other dictionary operations

- The **`in`** operator lets us see if a key is in a dictionary

```
if 'Spiderman' in superheroes:
  print('We have a webslinger!')
```

- You can also remove a key from a dictionary with the **`del`** operator

```
del superheroes['Spiderman'] # no more Spiderman!
```

Studying Advice

Studying advice

- Focus on quizzes
- Focus on assignments
- Memorizing things about Python is okay
- Practicing programming is better

- Hints:
  - You will probably have to do something with turtle
  - There might be some simple cryptography
  - Loops, `if` statements, strings, lists, and dictionaries are all fair game

Quiz

Upcoming

Next time…

- Exam 1!

Reminders

- Review chapters 1 through 4 of the textbook
- Work on Assignment 4