Introduction to Python

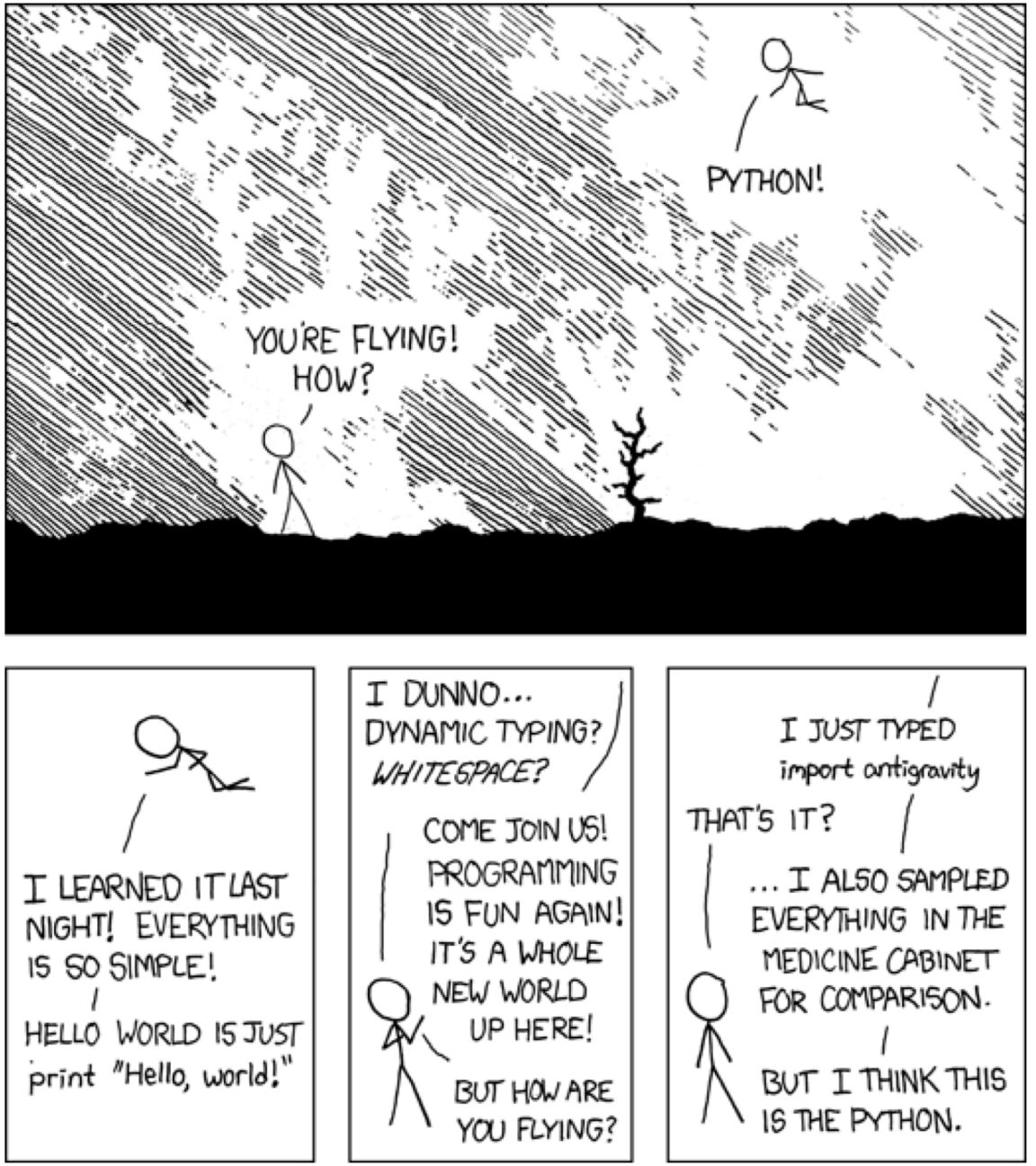
Ammon Perkes

BIOL 437 / GCB 536 / …

September 5, 2018

Overview

- 1. Using python
- 2. Variables and data types
- 3. Logical statements (if-else)
- 4. Loops
- 5. Lists
- 6. Application: String Matching



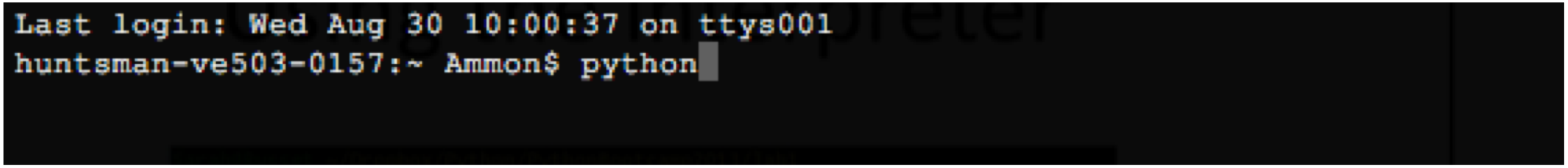
https://xkcd.com/353/

1. Using python: Three Approaches

- Use the Interpreter
  - Quickly see the results of a single line of code
  - Great for quickly testing small pieces
- Using scripts
  - Easily re-run the same code (important for longer blocks)
  - Share your code with others
- Interactive Notebooks like Jupyter or iPython
  - Best of both worlds, especially for demonstration
  - Easy learning curve, but requires some installation to set up
  - Text editing is not always as powerful as a text editor

1.1 Using python: The Interpreter

- Simply type ‘python’ in the terminal/command line
- The interpreter will **immediately** evaluate a line of code (unlike a script, which will not run until executed)
- It remembers variables entered, making it great for **testing small sections of code**. (or as a programmable calculator). For longer code, the interpreter quickly becomes difficult to work with.



1.1 Using python: Using the Interpreter

- Open your terminal
- Type python
- Type code and press enter to see the result

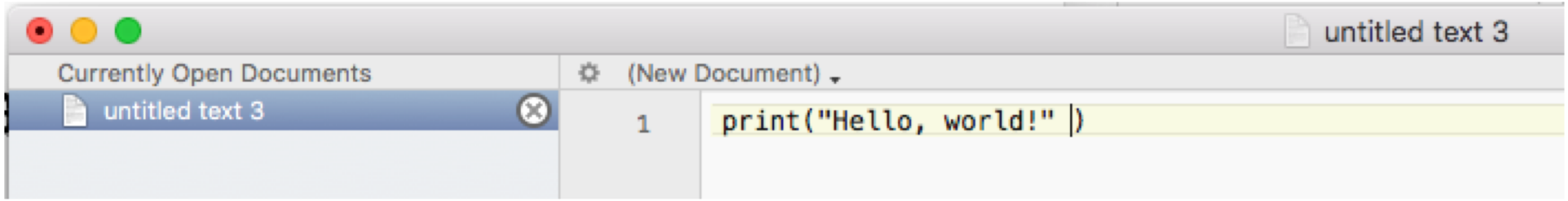
```
>>> print('Hello, world!')
Hello, world!
>>>
```

- Because it remembers variables, you can also use the interpreter as a programmable calculator

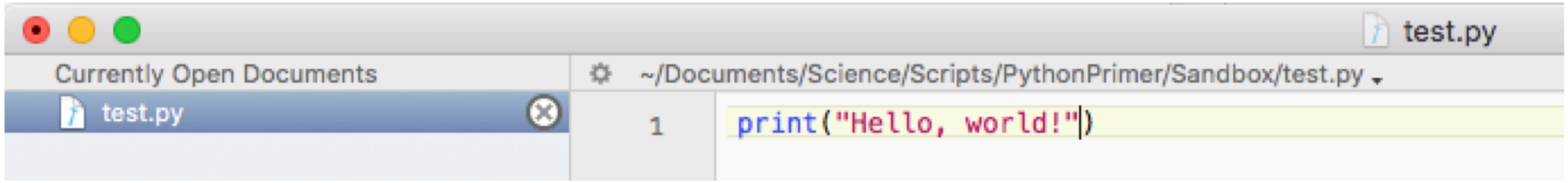
```
>>> x = 4
>>> y = 7
>>> z = x + y
>>> x + y + z
22
>>>
```

1.2 Using Python: Create and run a script

- Text editors make it easy to write, edit, and share code
- Step 1: Create a script
  - Open a plain text editor (Notepad++, BBedit, etc. MS Word won't work)
  - Type the following:



  - Save your file as test.py
  - Notice that, after saving, you likely have conditional highlighting



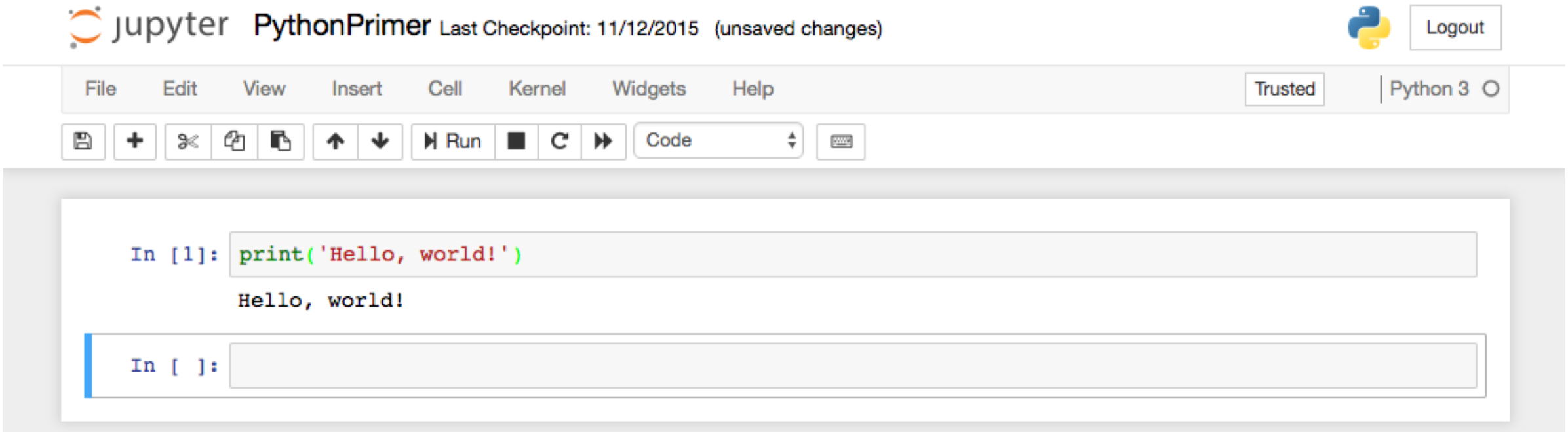
1.2 Using Python: Create and run a script

- Step 2: Running the script
  - Open terminal and move to the correct directory (using cd to change directory and ls or dir (for OSX or windows, respectively) to list files/folders in the directory.
  - One you have found your script, type python test.py
  - Assuming there are no errors in the code you should see your code's output:

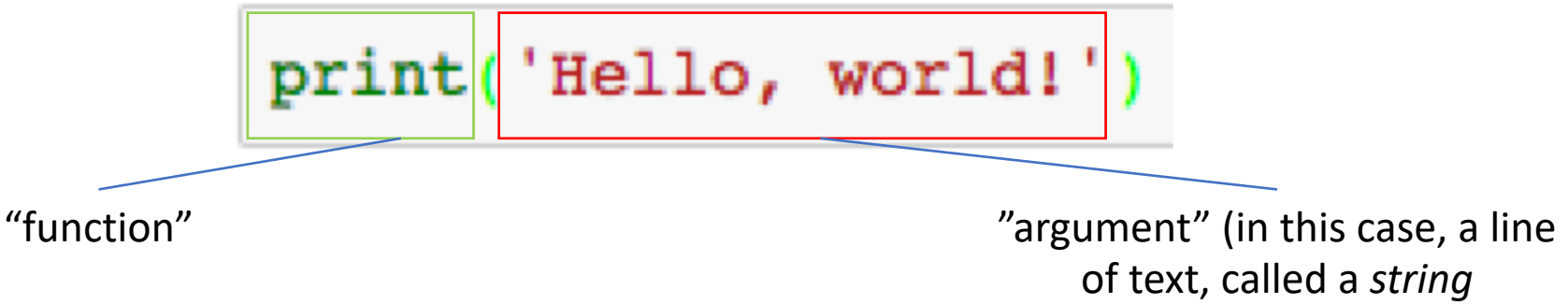
```
huntsman-ve503-0157:CompBio Ammon$ python test1.py
Hello, world!
huntsman-ve503-0157:CompBio Ammon$
```

1.3 Using Python: Interactive Notebooks

- Interactive notebooks like Jupyter or iPython can dramatically lower the learning curve and improve the experience of coding in Python
- They allow you to run code in sections, similar to the interpreter, but make saving, editing, and sharing your finished code much easier



1.4 Using Python: So what are we doing?



- This is called a *print statement*
- The print command outputs contents to the terminal screen
- To print a line of text (called a *string*), it must be enclosed in quotations. This identifies the text as a string instead of as some other command.
- Most commands are of the format command(arg)

1.4 Using python: fun with printing

- Try each of the following, in each one think about what print is doing

```
In [2]: print("Hello " + ",world!")
        print("Hello", "world")
        print(1,2,3)
        print(123)
        print(1+1)
        print(2*3)
        print("1+1")
        print("1" + "1")
```

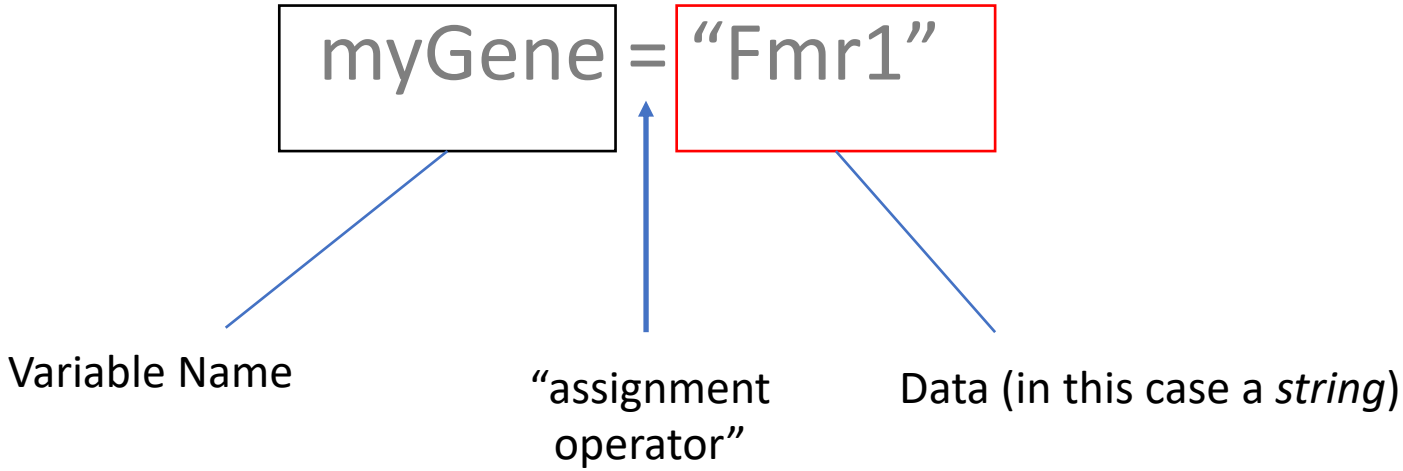
Tip: in terminal/command line you can press the up arrow to cycle through your history

2.1 Variables: Introduction

- You can think of variables as boxes for temporarily storing data. We name the boxes so that we can use them again later.
- Different languages operate differently, but variables must generally be *defined* (or *declared*) before being used. (Python is no exception, although it is quite flexible, making it comparatively easy)
- Variable names can be almost anything, but certain words (e.g. print) are reserved for commands.

2.1 Variables: Introduction

- For example:



This tells python: Store the value “Fmr1” in the box called myGene

2.2 Variables: Naming Rules and Conventions

- Rules (breaking these results in errors):
  - No spaces, no special characters. Use only letters, numbers, and underscores
  - Names cannot begin with a number (i.e. 7eleven won't work)
  - You can't use python-reserved words (i.e. print, int, if)
  - Names are case sensitive, so myGENE is a different variable than myGene
- Conventions (this makes code easier to read):
  - Begin a variable with a lower case letter
  - Make names descriptive (myGene instead of mg)
  - If using multiple words, use camelCase or under_scores to make it easier to read

2.2 Variables: good and bad examples

- Good variable names:
  - geneID
  - personCount
  - Input_file
  - avgGeneCount
- Bad variable names:
  - 3rdColumn
  - tnisfnld
  - person#
  - class

2.2 Variables: good and bad examples

- Good variable names:
  - geneID
  - personCount
  - Input_file
  - avgGeneCount
- Bad variable names:
  - 3rdColumn (illegal)
  - tnisfnld (gibberish)
  - person# (illegal)
  - class (reserved word)

2.3 Variables: Data types

- Data comes in different types (numbers, characters, words, etc)
- Variables work differently depending on the type of data they contain. We'll mostly be working with a few data types:
  - **String (**str**)**- a *string* contains text. You can think of it as a series of individual characters, like beads on a string. Strings are enclosed in quotes to distinguish them from variables or commands. ("double" and 'single' quotes both work)
  - **Integers (**int**)** – Just like in math, integers are counting numbers (1,2,3,-1, etc).
  - **Floating point numbers (**float**) –** numbers with decimals.
    Be aware Ints and floats sometimes work differently:
    (in Python 2, 7/10 != 7/10.0)
  - **Booleans (**bool**)** – True or False (1 or 0). Used for logical operations (discussed in detail later)

2.3 Variables: Data types

- Variable type is determined when the variable is defined
- So
  - myGene = “Fmr1” is a string
  - myGene = 8 is an int
  - myGene = 8.7 is a float
- if you redefine a variable it will change its type

2.4 Data Types: Boolean Variables

- A boolean (or “bool”) is a type of variable, just like strings and ints, but it can only take one of 2 values: True or False
- True and False are reserved words and must be capitalized
- Try not to think of them as words, they’re interpreted by the computer as 1 and 0, respectively. (In fact you can use 1 or 0 in place of True or False)

3.1 Logical Statements: if-else

- Think of programming as a way to give your computer (extremely specific) instructions
- Often we need to provide conditional instructions, e.g.
- To get to the bus stop by bike:
  - **If** it's before 6pm, stay north on 38th to chestnut, **otherwise** cut across campus after the crosswalk
  - When you get to chestnut, turn right, and continue until JFK.

3.1 Logical Statements: if-else

- In our instructions, the “if” “otherwise” provided a branch, depending on existing conditions.
- We do this in python using if and else
- In python, we would give the same instructions as

```
In [7]: daytime = True

print("Go west on Guardian Dr")
print("Turn right on University Ave")
if daytime:
    print("Continue straight on university for 3 blocks")
else:
    "After crosswalk, cut across the pedestrian path"
    "Continue diagonally past the love statue, the button, and starbucks"
print("Turn right on Chestnut")
print("Turn Left on JFK")
print("...")
```

True and False are special words called **Booleans**

This is the if/else statement. depending on the condition (daytime) it determines which section to execute

3.3 Logical Statements: Using if-else

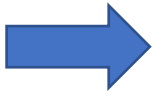
```
# How to use if-else statments
if conditional:
    'this code is executed'
else:
    'this other code is executed'

# Remember that your conditional must already be defined
```

- Note the colon following the *if* and *else* statements
- The spacing defines the *if* block (a good text editor is helpful here)

3.3 Logical Statements: Using if-else

- Anything that can be evaluated as true or false can be used
  - is a True?
  - is a or b True?
  - is a equal to b?
  - is a greater than or equal to b?
  - is b less than a and greater than c?
  - is a equal to “ACGTAT”?

3.3 Logical Statements: Using if-else



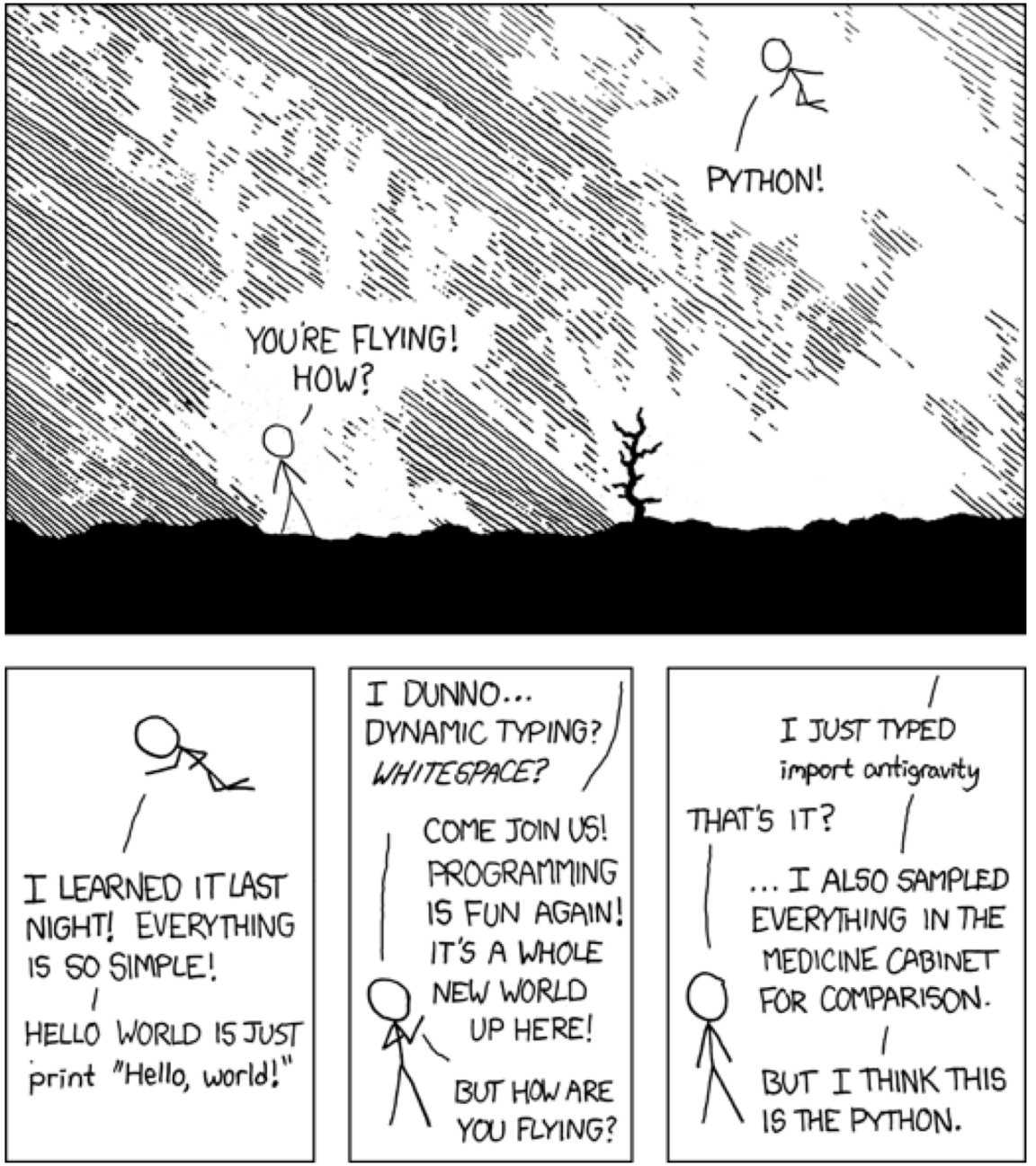
Symbol	Meaning	Example
==	is equal to	`if (a == 4):`
!=	is not equal to	`if (a != "applesauce"):`
<	is less than	`if (a < 10.5):`
<=	is less than or equal to	`if (a <= b):`
>	is greater than	`if (a > (b * 2)):`
>=	is greater than or equal to	`if (a >= -1):`
and	and	`if (a > 0) and (a != 3):`
or	or	`if (a > 5) or (a < -5):`
not	not	`if not (a == 1):`

5 Minute Break, next up: for and while loops!

```
while time < 90:
    if exhaustion == True or boredom == True:
        break
    else:
        continue
```

Overview

- 1. Using python
- 2. Variables and data types
- 3. Logical statements (if-else)
- 4. Loops
- 5. Lists
- 6. Application: String Matching



https://xkcd.com/353/

4.1 Loops: Intro to Loops

- Loops provide a tool to repeat an action multiple times
- Since programming almost always seeks to automate an otherwise exhaustive task, loops are one of the fundamental techniques
- There are only two loop operations:
  - the for loop
  - the while loop

```
for n in [0,1,2,3,4]:
    print(n)

n = 0
while n < 5:
    print(n)
    n = n + 1
```

4.2 Loops: The for Loop

- The for loop repeats an action a specified number of times
- **iterable**: anything you can work through in discrete steps (iterate). Strings, lists, files all can be iterated
- **var** takes on each value in the iterable (e.g. each letter in a string)
- When there are no values left, the loop ends.

```
for var in iterable:
    "do something"
```

```
for i in range(5):
    print(i)

for letter in "ATCAGTA":
    print(letter)
```

4.2 Loops: Using the for loop

- One of the simplest ways to use a for loop is with the range function

```
for i in range(5):
    print("Hello!")
```

- Any iterable can be used

4.3 Loops: The while Loop

- The while loop repeats an action an indefinite amount of times, based on some condition
- while loops are useful when you aren't sure how many times you will need to repeat an action
- Beware of creating infinite loops!
- If you're code doesn't seem to be doing anything (or if it's printing endlessly) hit CTRL-C to quit (or stop in Jupyter)

```
while condition:
    "do something"
```

```
count = 0
while count < 10:
    print(count)
    count = count + 1

answer = 0
while answer != "Ammon":
    answer = input("Guess what my name is: ")
```

4.3 Loops: Using the while loop

- Waiting for correct input from user
- Counting calculations (e.g. How many times can you divide 100 by 2?)
- Search until successful
- Anytime your loop length is unknown
- Watch out for endless loops! (Remember CTRL-C)

4.4 Loops: for vs while

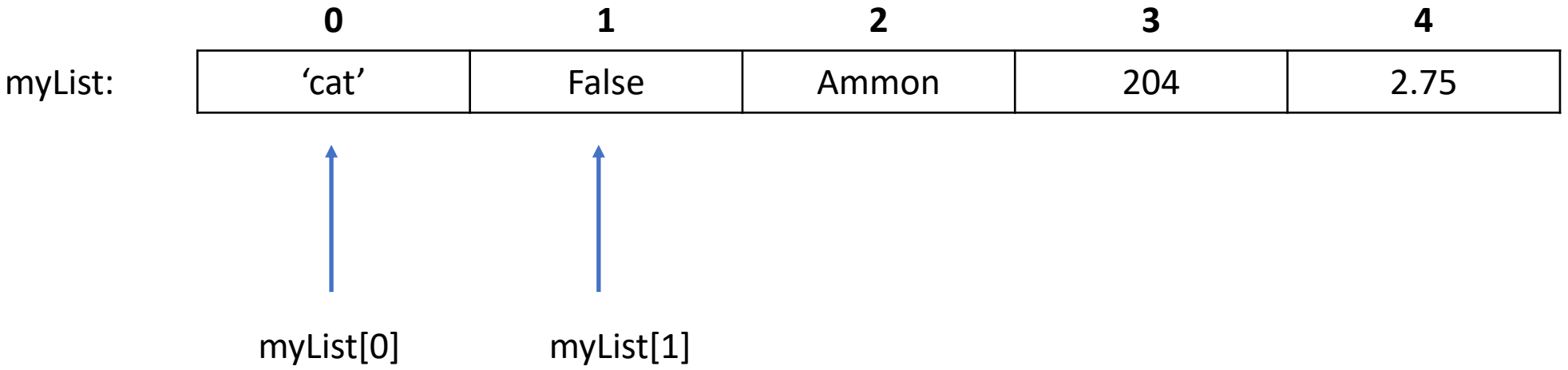
- Use for when:
  - number of steps is known
  - you want to go through all of the steps
- Use while when:
  - number of steps required is unknown
  - you can stop once the condition is met
- Loops often take the bulk of your run time, beware of inefficiencies within for loops.
- In while loops (but not in for loops) you can create an infinite loop

5.1 Lists: Intro to Lists

- A **list** is a built-in data structure in Python (i.e. a way of storing a series of data points)
- Lists are denoted with brackets (mylist = [“thing 1”, “thing 2”])
- Lists can be composed of anything (numbers, words, objects, other lists) and can be mixed ([1, “cat”, [1,2,3], False])
  - You can use any list in for loops

5.2 Lists: Using Lists

- Lists are indexed, starting at 0
- lists are accessed using listName[*index*]
- mylist = [“cat”, False, “Ammon”, 204, 2.75]



5.2 Lists: Creating lists

- Empty lists
  - myList = []
- List of elements:
  - myList = [“cat”, “fish”, “dog”]
- Quickly make lists of numbers using range:
  - myList = list(range(5,100,10))

5.3 Strings as lists

- Strings can be indexed and iterated just like lists
- myString = ‘This is a string’
- myString[5] = ’i’
- myString[0:6] = ‘This i’
- for letter in myString:
  - print(letter)

7.1 Learning Python

- Practice!
- Read supplemental materials
- Use Google, watch YouTube videos, ask real humans for help
- Notes on debugging
  - Read your error message, note the line that’s causing problems
  - Google (you can google your error message for explanations on why it broke)
  - Use print to check values before it breaks
  - Walk through code in interpreter
  - Write out code
  - Use proper debugging tools (pbd, others)

6.1 Application: String Matching

- Check if some arbitrary section is found in your reference DNA
  - refSeq = "ATACGGCAATATGACGCATTCGTGAGGCATACGACG”
  - querySeq = "GAGGCATA”

```
### String Search ###
# Write a program which checks if some arbitrary string is in your sequence

refSeq = "ATACGGCAATATGACGCATTCGTGAGGCATACGACG"
querySeq = "GAGGCATA"

refLength = len(refSeq)
queryLength = len(querySeq)

matchCount = 0
for i in range(refLength - queryLength + 1):
    matchedPositions = 0
    for nt in range(queryLength):
        if querySeq[nt] == refSeq[i+nt]:
            matchedPositions = matchedPositions + 1
    if matchedPositions == queryLength:
        print "Match found at position " + str(i)
        matchCount = matchCount + 1
if matchCount == 0:
    print "No match found :("
else:
    print matchCount,"total matches found!"
```