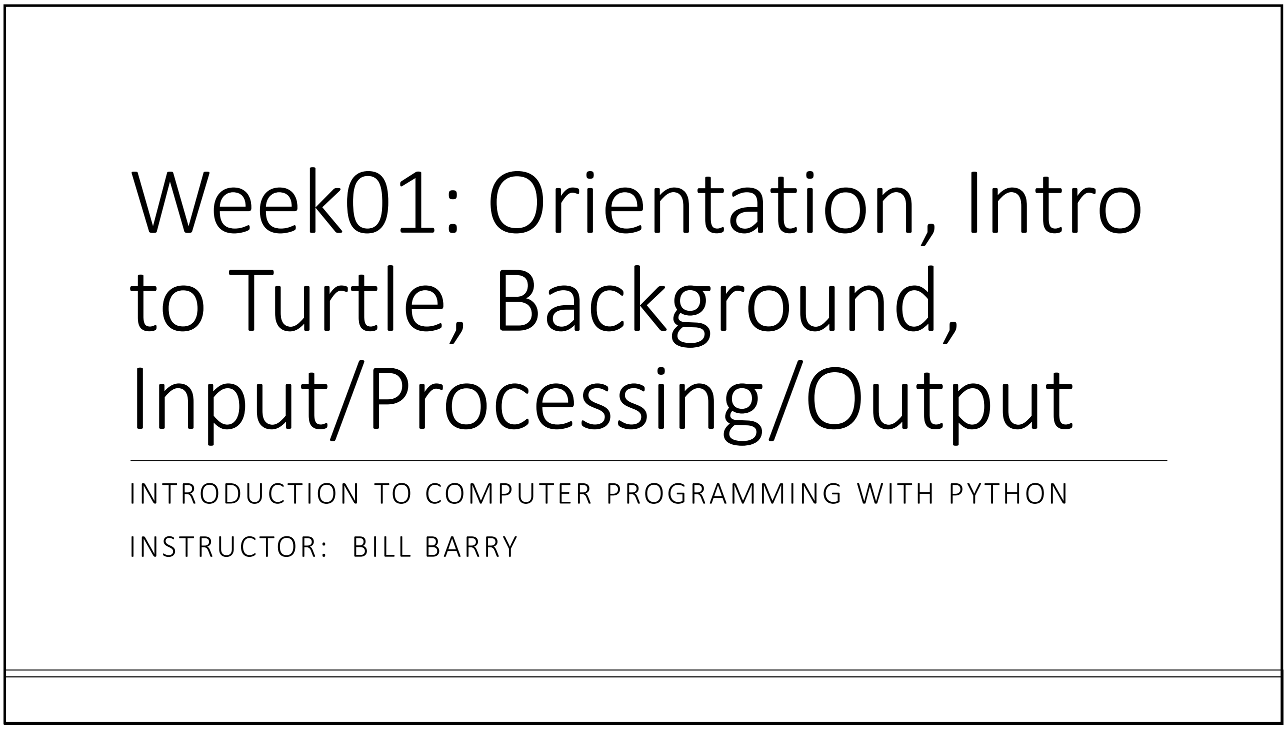
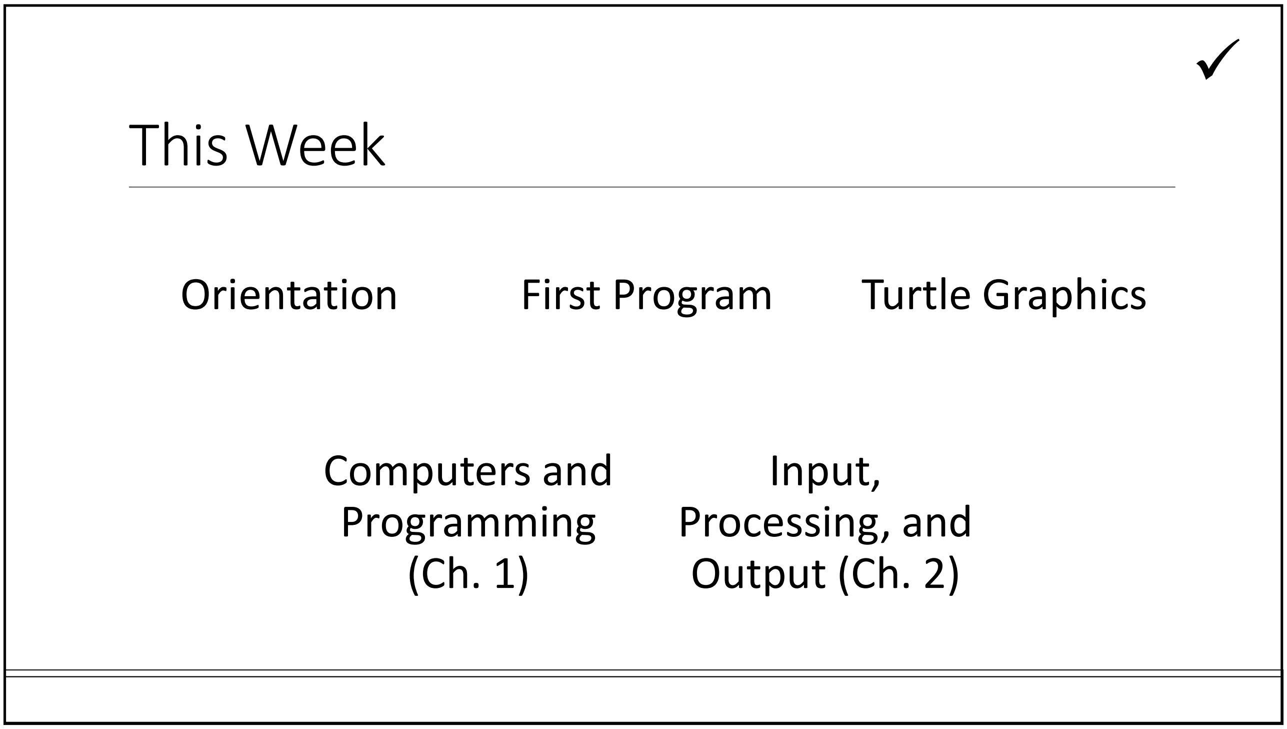


Week01: Orientation, Intro to Turtle, Background, Input/Processing/Output

INTRODUCTION TO COMPUTER PROGRAMMING WITH PYTHON

INSTRUCTOR: BILL BARRY

1



✓

This Week

Orientation

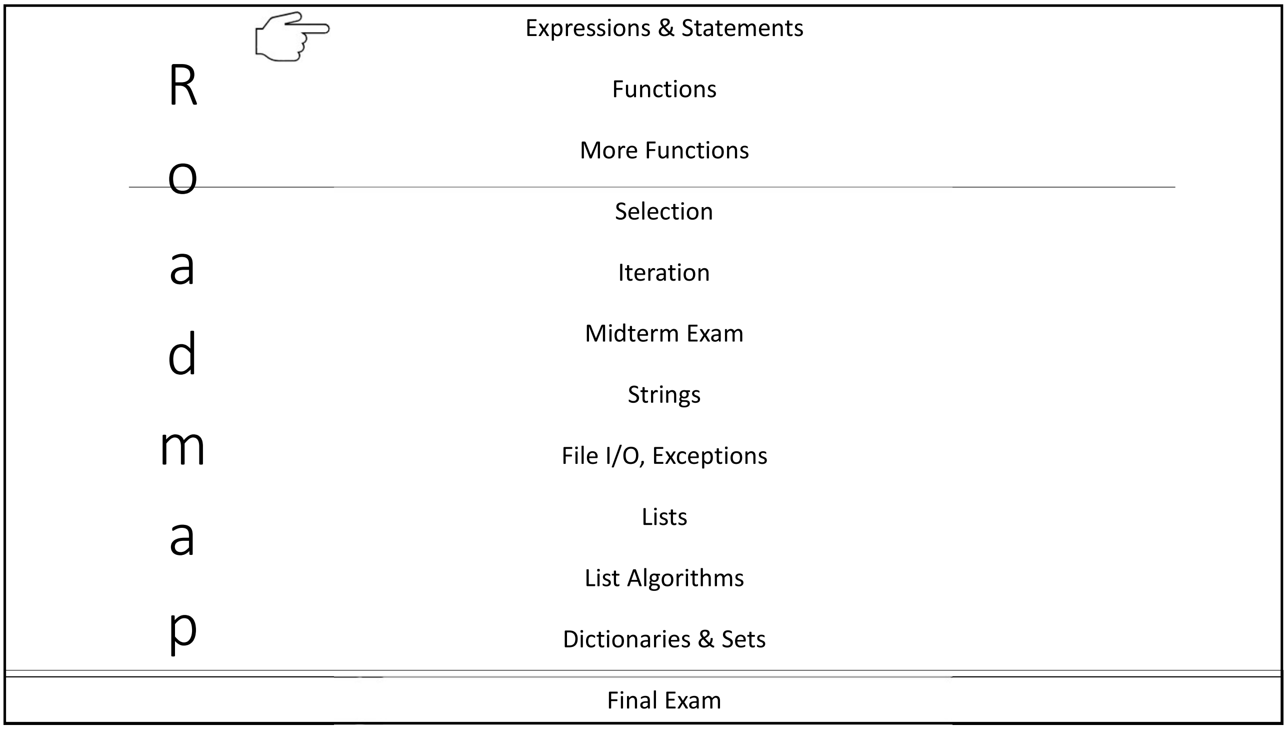
First Program

Turtle Graphics

Computers and Programming (Ch. 1)

Input, Processing, and Output (Ch. 2)

2



Roadmap

☞ Expressions & Statements

Functions

More Functions

Selection

Iteration

Midterm Exam

Strings

File I/O, Exceptions

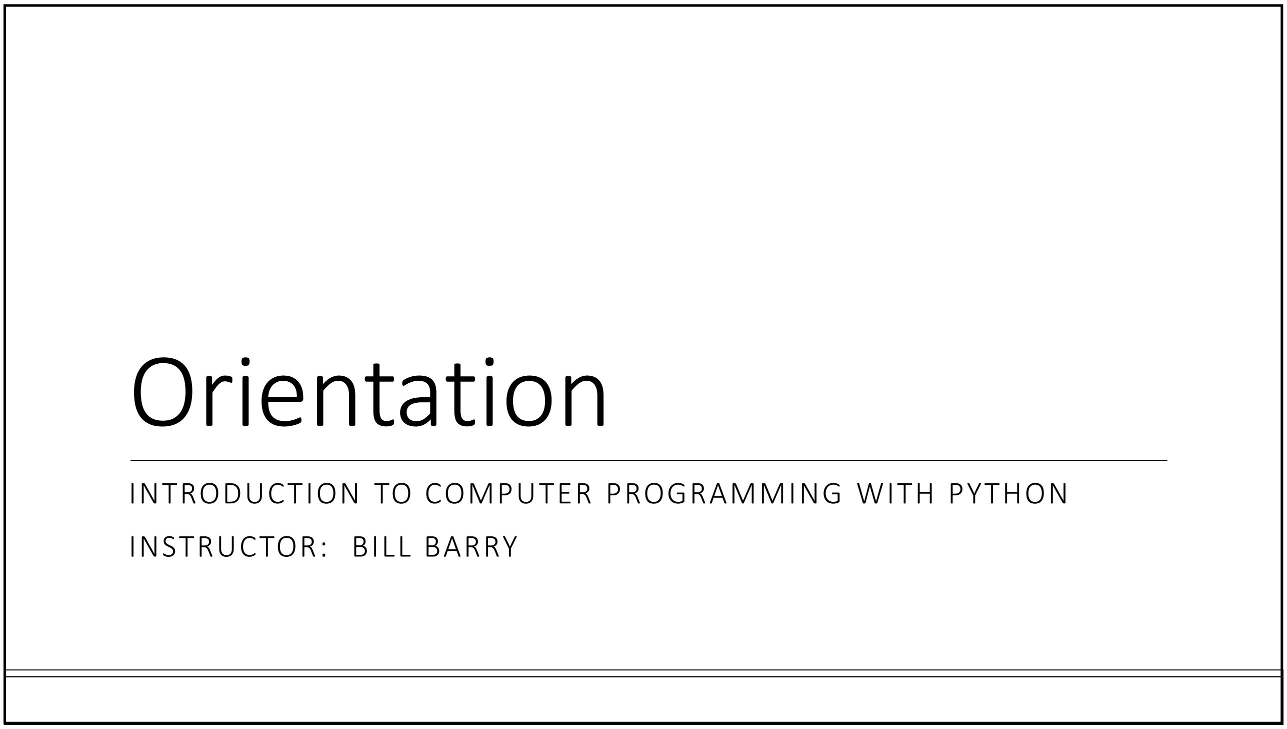
Lists

List Algorithms

Dictionaries & Sets

Final Exam

3



Orientation

INTRODUCTION TO COMPUTER PROGRAMMING WITH PYTHON

INSTRUCTOR: BILL BARRY

4

What We'll Cover

About your instructor

Expectations for in-person courses

Syllabus and Schedule

Course content

Course website

Basic concepts of a computer program and programming

The Python language

5

About Your Instructor (Bill Barry)

B.S. in Computer Science/Business in 1985 (The University of Texas at El Paso)

Masters in Software Engineering in 1995 (Seattle University)

Technical career mostly at Microsoft starting at 1991
- Senior Lead Software Engineer
- Focused on Software Testing and Tester Education
- Last stint ended in May 2015

Started programming in junior high; got really into it in high school

Teaching computer-related topics since 1983
- My first-grade teacher said I was teaching and tutoring even then
- Now full time, tenure track at North, teaching Computer Science

Fun stuff
- Biggest non-computer hobbies: music, baking
- Languages: fluent in Spanish; bit of ASL, German, French, Japanese

6

Virtual Course: Making It work

Sessions

- Be present
- Be prepared
- Be inquisitive

Zoom

- Be personal
- Be engaged
- Be cooperative

Canvas

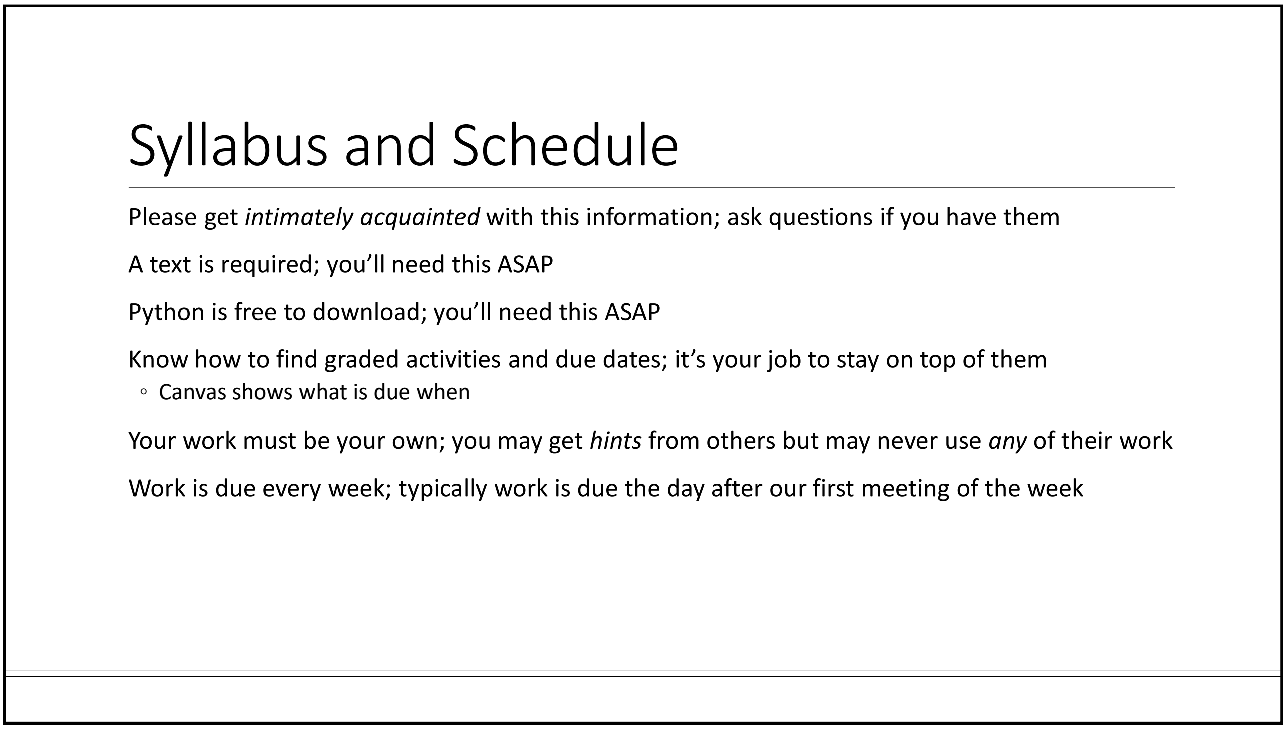
- Be vigilant
- Be timely
- Be proactive

7

Expectations for Virtual Courses

- Show up as often as you can, as punctually as you can; be engaged in Q&A
- Read your text and supplemental materials
- Study/learn from posted examples
- Work through weekly lab activities; take weekly quizzes
- Write programs by yourself (your own code); cite any references used (web or human), especially for things substantially different from in-class approach
- Take exams online via Canvas with no help from other humans
- Stay in contact; if you're getting off track, it's best to say it sooner vs. later
- Fill out a course evaluation at the end of the course
- Participate in forum discussions (recommend: subscribe to each)

8



Syllabus and Schedule

Please get *intimately acquainted* with this information; ask questions if you have them

A text is required; you'll need this ASAP

Python is free to download; you'll need this ASAP

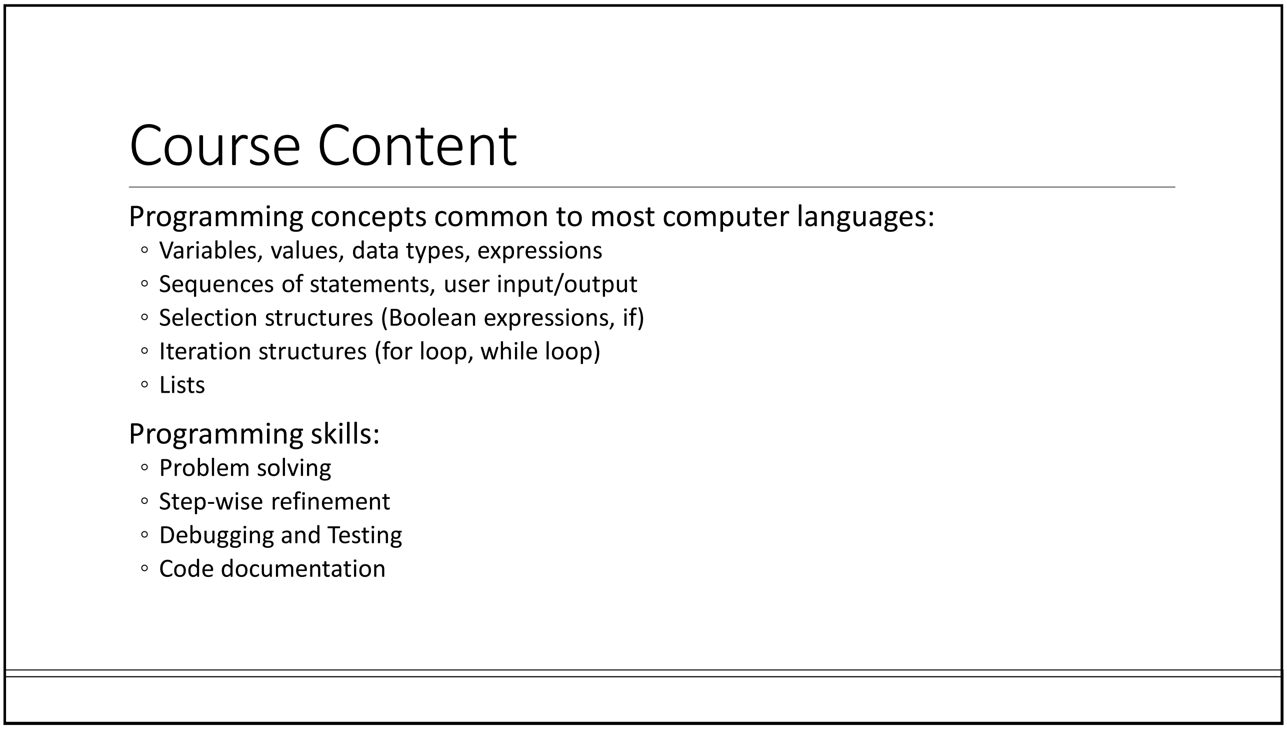
Know how to find graded activities and due dates; it's your job to stay on top of them

- Canvas shows what is due when

Your work must be your own; you may get *hints* from others but may never use *any* of their work

Work is due every week; typically work is due the day after our first meeting of the week

9



Course Content

Programming concepts common to most computer languages:

- Variables, values, data types, expressions
- Sequences of statements, user input/output
- Selection structures (Boolean expressions, if)
- Iteration structures (for loop, while loop)
- Lists

Programming skills:

- Problem solving
- Step-wise refinement
- Debugging and Testing
- Code documentation

10

Course Website on Canvas

Basics

- Course is run from this site: http://canvas.northseattle.edu/
- Know this backup in case of website trouble at North: http://northseattle.instructure.com
- You're automatically added

Key sections

◦ Home	access most things you'll need
◦ Syllabus	see the complete syllabus and schedule
◦ Grades	review your progress; test "what if" scenarios
◦ Calendar	see when work is due

Typical week contents

- Materials
- Discussion Forum
- Assignments: one lab (usually on CodingBat), one quiz, one project

Let's look at how to…

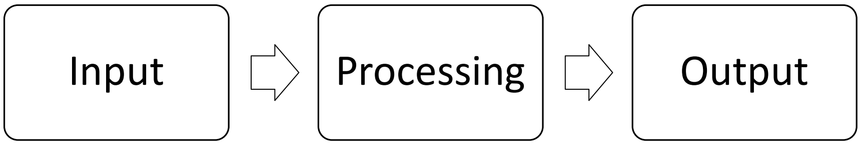
- See the calendar
- Submit work via the site

11

Computer Programs

What is a program? It is a set of ordered instructions written in a programming language that, when followed (aka "executed"), performs a particular action

Another way of viewing a typical computer program:



Input

Processing

Output

12

Computer Programming

What is computer programming?

- **Planning** your work, especially with hard/interesting problems
- **Coding** the sequence of steps that the computer will follow to solve a problem
- **Testing** the code to see if it works properly in “happy path” as well as unexpected cases
- **Documenting** your work
- *Note: modern practices don’t separate these activities cleanly, and may sequence them differently*

To become a good programmer requires:

- **Thinking logically** about problems and **breaking them down** into appropriate pieces
- Good **knowledge of the tool** (computer language) being used including an understanding of what can (and cannot) be done with it
- Lots of **practice**!

If you like solving puzzles, it can be highly satisfying and enjoyable, too ☺

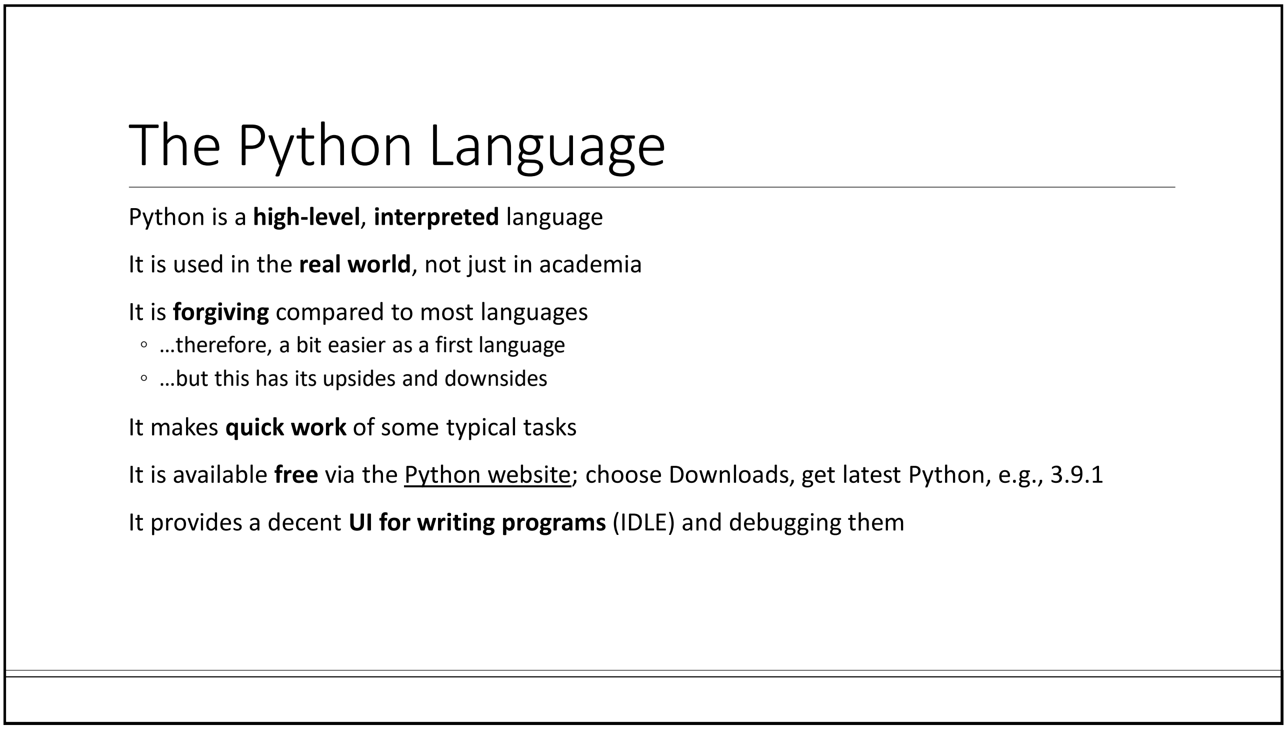
13

Programming and Creativity: Analogy

More like this…

…than this (at least the instruction part)

14



The Python Language

Python is a **high-level**, **interpreted** language

It is used in the **real world**, not just in academia

It is **forgiving** compared to most languages

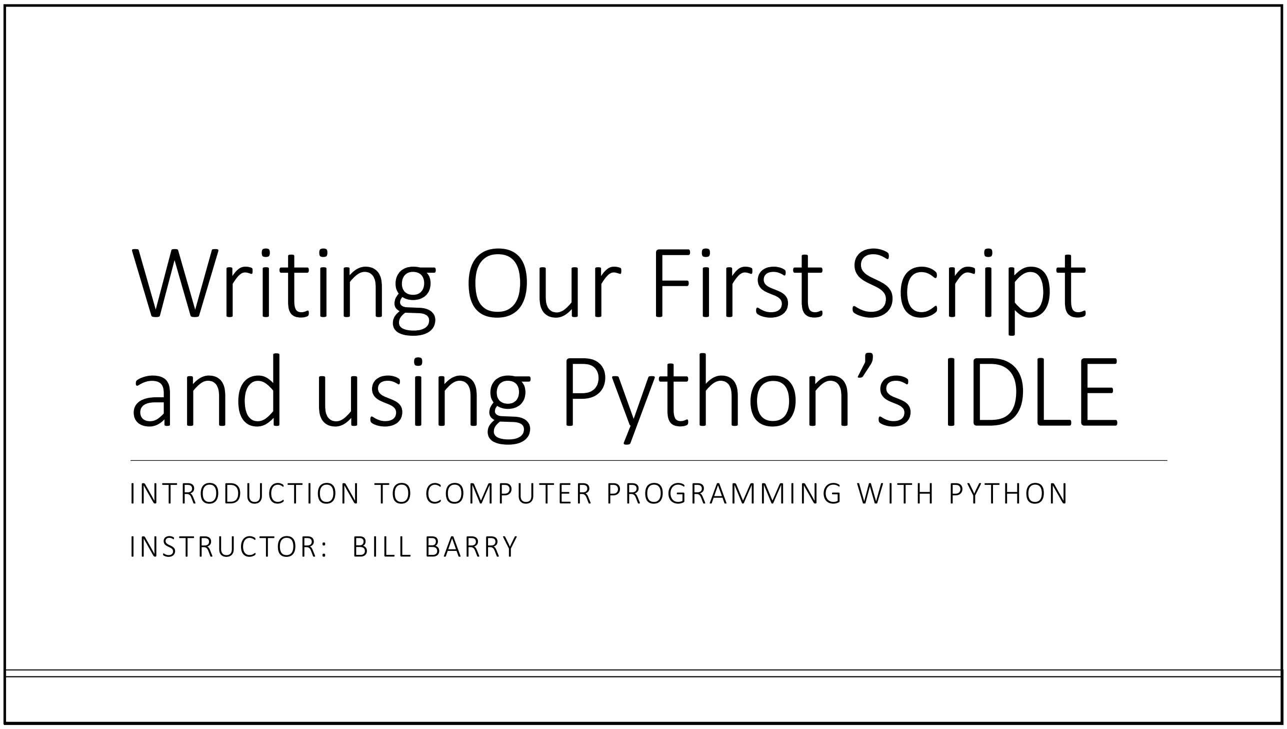
- …therefore, a bit easier as a first language
- …but this has its upsides and downsides

It makes **quick work** of some typical tasks

It is available **free** via the Python website; choose Downloads, get latest Python, e.g., 3.9.1

It provides a decent **UI for writing programs** (IDLE) and debugging them

15



Writing Our First Script and using Python's IDLE

INTRODUCTION TO COMPUTER PROGRAMMING WITH PYTHON

INSTRUCTOR: BILL BARRY

16

The Python IDLE

IDLE = **I**ntegrated **D**eve**L**opment **E**nvironment

Has an interactive mode where you can “play”

- Good to know and use
- …but it’s **not where we create our project work** IMPORTANT NOTICE

Also lets you create and edit files

- These have an extension of .py
  - If on Windows, please make sure you have file extensions showing!
- This is what we’ll use to write/edit
- You’ll submit these files for your labs and projects

Let’s check out IDLE and write our first simple program

17

Turtle Graphics

INTRODUCTION TO COMPUTER PROGRAMMING WITH PYTHON

INSTRUCTOR: BILL BARRY

18

Turtle Graphics: Coordinates

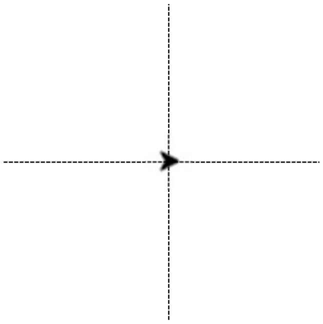
Python includes Turtle Graphics, a fun way to get started with programming. This idea originated with the programming language Logo. While we won’t use this for assigned projects, it’s a great way to illustrate basic concepts and is *fun*

Basic Concepts

Graphics work in a cartesian coordinate system, ~800 x 800 units

The “turtle” starts at (0,0), facing right

As you move the turtle, it leaves a trail; use this to draw lines



19

Turtle Graphics: Programming Basics

Turtle Programming Basics

At the start of your program, write this:

```
from turtle import *
```

At the end, write this:

```
done()
```

In the middle, write whatever commands accomplish the desired task

Important: **make sure you don’t create a file named turtle.py** – if you do, it tries to import turtle commands from that file instead of the official Python Turtle library, which isn’t a good thing!

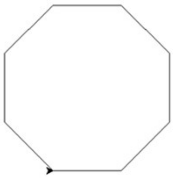
20

Turtle Graphics #1

Let’s learn a few basic commands

Command	Description	Example
color	Sets the color of the drawn line	`color('blue')`
forward	Moves the turtle forward the specified number of steps	`forward(30)`
left	Turns the turtle left by the specified number of degrees	`left(20)`

Challenge #1: Draw an Octagon (100 units per side)



There are lots of named colors; you can also specify them by RGB or hex value. See this site for a list of named colors: https://ecsdtech.com/8-pages/121-python-turtle-colors

Challenge #1 Answer

```
from turtle import *
forward(100)
left(45)
forward(100)
left(45)
forward(100)
left(45)
forward(100)
left(45)
forward(100)
left(45)
forward(100)
left(45)
forward(100)
left(45)
forward(100)
left(45)
done()
```

Whew, there has **got** be a better way to repeat stuff, huh? Stay tuned…

Turtle Graphics #2

Let’s learn a few more commands:

Command	Description	Example
color	Sets the color of the drawn *line* and *fill*	`color('green', 'yellow')`
begin_fill	Issued *before* you draw, prepares for later filling of the drawn shape	`begin_fill()`
end_fill	When finished drawing, this command will fill the drawn shape	`end_fill()`
pensize	Specifies the thickness of a drawn line; default is 1	`pensize(5)`

Challenge #2: Draw a Stop Sign (black outline, red fill, pen size of three, no letters)

23

Challenge #2 Answer

```
from turtle import *
color('black', 'red')
pensize(3)
begin_fill()
# ...rest of octagon code here...
end_fill()
done()
```

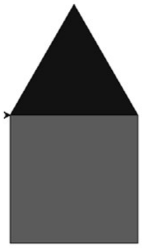
24

Turtle Graphics #3

And even more commands:

Command	Description	Example
goto	Moves the turtle to the specified position, without changing orientation	`goto(-50, 25)`

Challenge #3: Draw a green house (200 steps per side) with a blue roof



25

Challenge #3 Answer

```
from turtle import *
# House
color ('black', 'green')
begin_fill()
forward(200)
left(90)
forward(200)
left(90)
forward(200)
left(90)
forward(200)
left(90)
end_fill()
```

```
# Roof
goto(0, 200)
begin_fill()
color('black', 'blue')
forward(200)
left(120)
forward(200)
left(120)
forward(200)
left(120)
end_fill()
done()
```

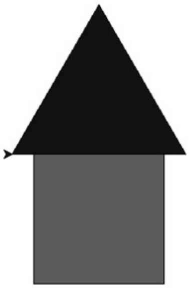
26

Turtle Graphics #4

And yet more commands:

Command	Description	Example
up	Picks up the turtle; often used before repositioning, so lines aren’t drawn	up()
down	Puts down the turtle	down()

Challenge #4: Draw a green house with a blue roof, with some eve overhang



27

Challenge #4 Answer

```
from turtle import *
## House
color ('black', 'green')
begin_fill()
forward(150)
left(90)
forward(150)
left(90)
forward(150)
left(90)
forward(150)
left(90)
end_fill()
```

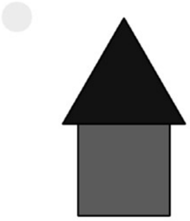
```
# Roof
up()
goto(-25, 150)
down()
begin_fill()
color('black', 'blue')
forward(200)
left(120)
forward(200)
left(120)
forward(200)
left(120)
end_fill()
done()
```

28

Turtle Graphics #5

And the last set of commands for today:

Command	Description	Example
circle	Draws a circle of the specified radius	`circle(50)`



Challenge #5: Draw a blue house with a green roof, with some eve overhang, and the sun in the sky

29

Challenge #5 Answer

```
from turtle import *
## House
color ('black', 'green')
pensize(3)
begin_fill()
forward(150)
left(90)
forward(150)
left(90)
forward(150)
left(90)
forward(150)
left(90)
end_fill()
```

```
# Roof
up()
goto(-25, 150)
down()
begin_fill()
color('black', 'blue')
forward(200)
left(120)
forward(200)
left(120)
forward(200)
left(120)
end_fill()
```

```
# Sun
up()
goto(-100,300)
begin_fill()
color('yellow')
#color('#FFD700')
circle(25)
end_fill()
hideturtle()
done()
```

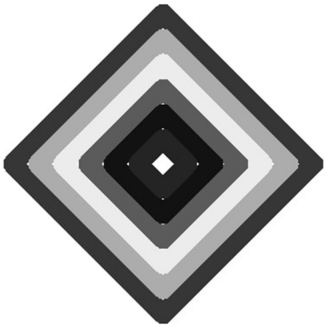
30

Turtle Graphics #6

Using what you've learned today, draw the figure shown at right

Use *only* what you've learned so far in the class

There are inefficiencies you should see and acknowledge; we'll learn to solve those as we move through the course



31

Challenge #6 Answer

```
from turtle import *
pensize(20)

color('red')
up()
goto(-150,0)
down()
left(45)
forward(212)
right(90)
forward(212)
right(90)
forward(212)
right(90)
forward(212)
right(90)

color('orange')
up()
goto(-125, 0)
down()
forward(177)
...

color('yellow')
up()
goto(-100, 0)
down()
forward(141)
...

color('green')
up()
goto(-75, 0)
down()
forward(106)
...

color('blue')
up()
goto(-50, 0)
down()
forward(71)
...

color('purple')
up()
goto(-25, 0)
down()
forward(35)
...

up()
goto(-175, 0)

done()
```

32

The Point of All This

Programming can be satisfying and fun; it's like solving a puzzle

Programming is about…

1. Thinking in a logical and orderly fashion
2. Breaking down a big problem into smaller problems
3. Understanding the basics of the commands, and using them properly
4. Coding, testing, iterating until you have a solution that works

33

Computers and Programming (Ch. 1)

CSC110: INTRO TO COMPUTER PROGRAMMING WITH PYTHON

INSTRUCTOR: BILL BARRY

34

What We’ll Cover

The chapter contains much more, but here we’ll touch on…

Hardware and software

Storing data

Syntax and keywords

Compilers and interpreters

35

Hardware and Software

Hardware

CPU, RAM (main memory), secondary storage device(s), input devices, output devices

Software

System software: operating system/OS, utility programs, software development tools

Application software: word processing, spreadsheets, email clients, web browsers, games

36

Storing Data

Why learn *binary* (base 2)?

- Something computer programmers need to know
- Useful in understanding minimum and maximum numbers given storage space; for example
  if you have eight bits (one byte) of storage, max is 11111111, so range is 0 to 255 (256 distinct values)

How do I convert from decimal to binary and vice-versa?

- Book has lots of visual examples
- To check your work, use programmer view in your computer's calculator program

Cool things you can do in binary

- Multiply by two → shift left one position: 0110 x 2 = 1100 (6 x 2 = 12)
- Divide by two → shift right one position: 1010 / 2 = 0101 (10 / 2 = 5)

Another common number base we should know is *hexadecimal* (base 16)

- Expresses more data in fewer digits; four binary digits become one hex digit

37

Using Python to Convert from Decimal to Binary (base 2), Octal (8), Hexadecimal (16)

If you want to play with number bases, there's an easy way in Python to convert from decimal to other common computer number bases

The example at right uses IDLE to demonstrate the bin(), oct(), and hex() functions. You can also do this in a program using the print() function

You can convert back by simply typing in the prefixed non-decimal number and pressing <Enter>, or using the int() function

```
>>> bin(29)    # convert decimal 29 to binary
'0b11101'
>>> oct(29)    # convert decimal 29 to octal
'0o35'
>>> hex(29)    # convert decimal 29 to hexadecimal
'0x1d'
>>> 0o35       # show value of octal 35 in decimal
29
>>> int(0x1d)  # convert hexadecimal 1d to decimal
29
>>>
```

Note the prefixes: zero (0) followed by a letter, then the non-decimal number

38

How Programs Work

Programs are usually written in a human-readable (high-level) language

- The code we write code is called "source code"
- Examples of high-level languages: C/C++, C#, Java, JavaScript, Python, Ruby

But the CPU can only run programs that are native to it (machine code)

How do we get from source to machine code?

There are a few different paths

39

Compilers and Interpreters

Analogy: written speech translation vs. real-time interpreter

Code must eventually be translated from human-readable form into *machine instructions*

Pure **Compilers** take entire source programs (e.g., C's .c or C++'s .cpp files) and convert them into a machine-executable binary. To run the program, you run the executable (.exe)

- To share your programs, you share only the exe; the source code stays with you
- Syntax errors are found during the compile step. You know that once you have an .exe, *that* type of error has been flushed out

Interpreters take a line of code from a source program (e.g., .py), convert it into binary, and execute the resulting code. Then they do the same for the next line. JavaScript is another example

- To share your programs, you share the source code; the user needs an interpreter on her end
- Syntax errors are found when the source code is interpreted; you must attempt to run that line of code before you know if a syntax error was found in it

There are hybrids. Java "compiles" into a theoretical machine language (JVM), with the result called "byte code;" this still requires further interpretation to run on a specific CPU. The .NET languages are other examples

40

Syntax and Keywords

Programming languages have a syntax. Spoken languages have syntax, too.

English, oversimplified: noun verb noun, understood as subject verb object

This is valid and correct; we understand it and get the correct mental picture:
the boy throws the ball

This is a *syntax error*; we don’t understand what it means:
throws the boy the ball

This is a *logic error*; it is syntactically correct but doesn’t express the idea correctly:
the ball throws the boy

Computer languages do not deal with ambiguity; we must express ourselves via correct syntax

41

Syntax and Keywords

Keywords are terms that are built into the language and are understood by it natively, without our assistance or explanation

In Python, examples include:

- `if    and    while    True    not`

When you use the IDLE, you’ll see these terms colored in a specific way:

```
if age > 20 and validLicense == True:
```

One implication of keywords is that when we get to make up our own names for things, we cannot use these terms

42

What’s Required for Programming?

Logic

Puzzle Solving

Detail+ Syntax

Programming: making computers do what we want; creating software to run on hardware

Thinking logically about how to solve a problem; turning that into detailed steps

More about thinking than about typing; the bigger the problem, the more design needed

Fixing the inevitable syntax and logic errors

Writing straightforward, maintainable code

Testing and refining the code

43

Input, Processing, Output (Ch. 2)

CSC110: INTRO TO COMPUTER PROGRAMMING WITH PYTHON

INSTRUCTOR: BILL BARRY

44

What We’ll Cover

The chapter contains much more, but here we’ll touch on…

Program development cycle, design aids (pseudocode and flowcharts)

Input, processing, and output (input, print)

Variables: naming, assignment, data types, literals

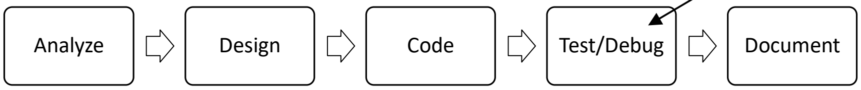
Calculations, operators, precedence and parentheses

Formatting: data types, examples, reference

45

Program Development Life Cycle

“Waterfall Model” Overview:



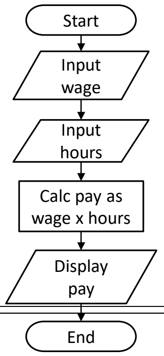
What’s the difference?

During design phase you might use…

Pseudocode:

```
Get wage from user
Get hours worked from user
Calculate pay amount:  wage * hours
Display pay amount
```

Flowchart:



Note: “design” will evolve as programs get complex; higher-level design concepts will be added

46

Input, Processing, and Output

Get data from the user using: `input`

Display data to the user using: `print`

Both must be followed by parentheses

I recommend you use quotes; most other languages want those around text

Use apostrophes or quotes around literal text you want to display, e.g.,

- This is wrong: `print(Hello)` (Python thinks, “What is this Hello thing?”)
- This is right: `print("Hello")` (Python thinks, “Display this exact text”)

Result of user input needs to be stored somewhere, so we put that on the left, e.g.,

- `userName = input("Enter your name: ")` Read as: ask user for her name and store the result in a place in memory I’ll refer to as *userName*

47

Escape Characters

Sometimes when you print (display on the screen), you need extra help

Most languages provide some special characters you can use here; Python is no different

Here are some key “escape characters” and their meanings:

Escape Character	**Meaning**
\t	Horizontal Tab
\n	Linefeed (newline, LF)
\'	Apostrophe (')
\"	Quotation Mark (")
\\	Backslash (\)

Example:

```
print("\tHello\n\tWorld")
```

Output:

```
        Hello
        World
```

Note: tabs can be used for alignment, though we’ll learn something better soon

48

Variables: An Analogy

A variable is like a cubbyhole in computer memory

Each cubby has a name label; on it, you write a name that describes the intended contents

- This is *defining* the variable

You place into the cubby data that you need to keep around for a while

- This is *assigning* the variable

Later, when you mention the cubby's label, Python will grab out the current contents and give them to you

For example, you can mention *scarf* and voila! the scarf is in your hands and ready to use

49

Variables

Data you want to hold in computer memory is kept in a *variable*

Names start with a letter and include letters, numbers, underscores, but NOT spaces

- Bad: `user age` `1stPayment` `miles-per-gallon`
- Good: `userAge` `payment1` `miles_per_gallon` or `milesPerGallon`

Assigning values to variables takes the form: *variable = expression*

- Don't confuse this with *equality*; it's not!
- Valid: `userAge = 29` `userAge = userAge + 1`

Note: case matters! **userAge** is not the same variable as **userage**

Data can be strings, integers, floating point numbers:

- String literals need apostrophes or quotes; numeric literals don't
- Examples: `age = 29` `taxRate = 0.0925` `empName = "Dana"`

50

Using Variable Contents

You can use variables anywhere you can use a literal; the name summons the content

Example:

```
print("Hello")
print("Bill")
```

…could become…

```
userName = input("Enter your name: ")
print("Hello")
print(userName)
```

…or…

```
userName = input("Enter your name: ")
print("Hello", userName)
```

The parens for print contain multiple items; use commas between them. You get spaces between, for free

Inputs From User are *Strings*

When users type things, they come in as strings (text), not numbers

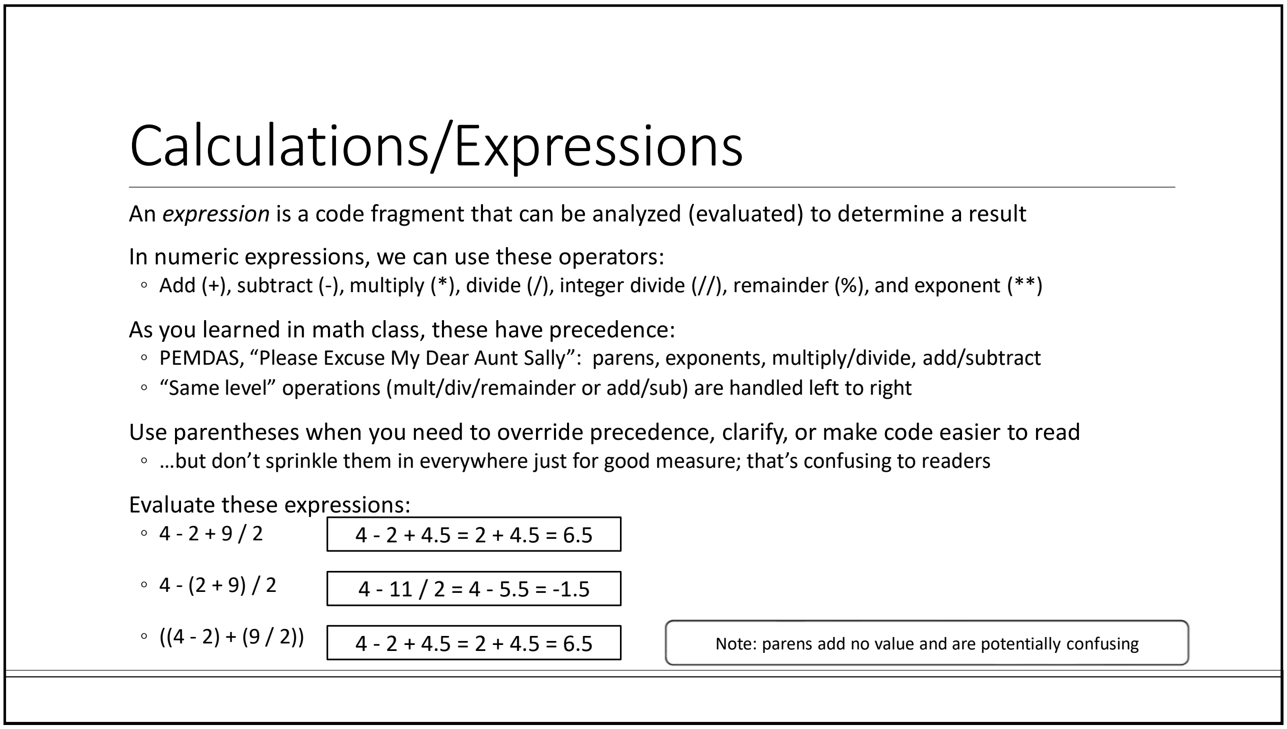
If we want them to be numbers, we convert them using int() or float()

Example (blows up):

```
userAge = input('Enter your age: ')
userAge = userAge + 1           # birthday
TypeError: Can’t convert 'int' object to str implicitly
```

Example (works):

```
userAge = input('Enter your age: ')
userAge = int(userAge)
userAge = userAge + 1           # birthday
```



Calculations/Expressions

An *expression* is a code fragment that can be analyzed (evaluated) to determine a result

In numeric expressions, we can use these operators:
- Add (+), subtract (-), multiply (*), divide (/), integer divide (//), remainder (%), and exponent (**)

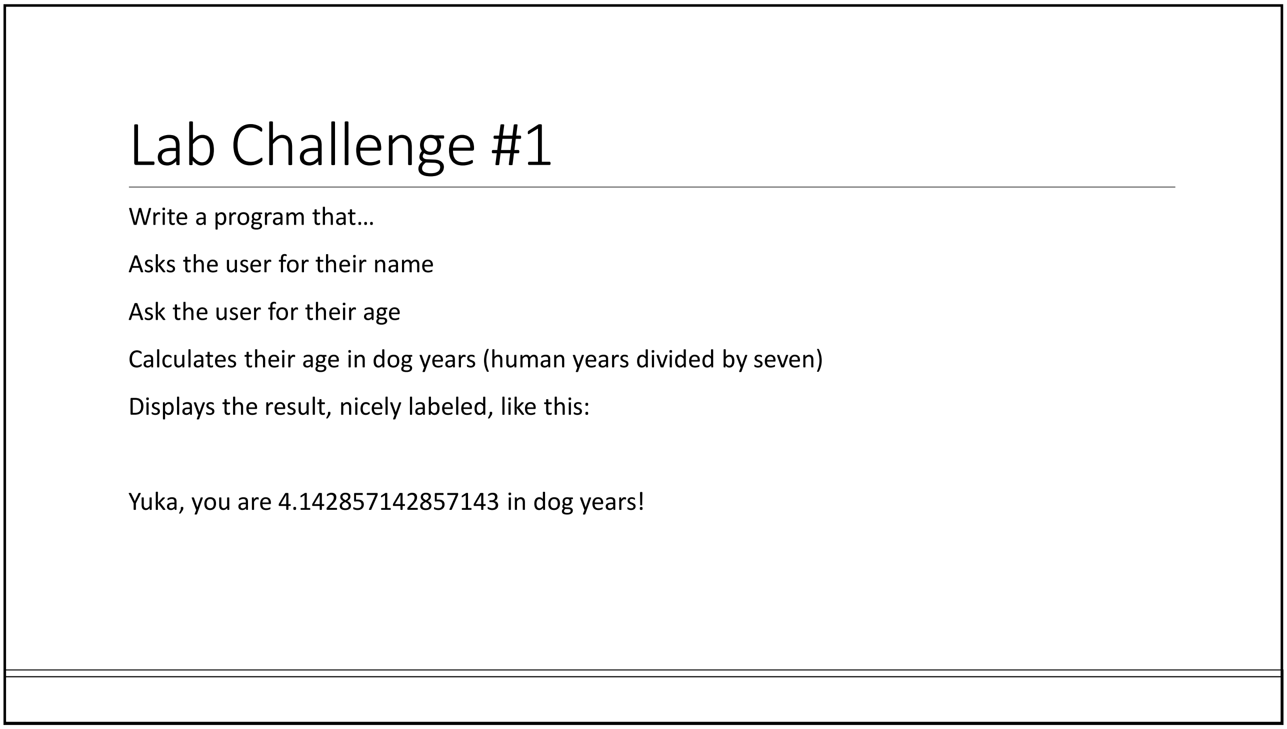
As you learned in math class, these have precedence:
- PEMDAS, “Please Excuse My Dear Aunt Sally”: parens, exponents, multiply/divide, add/subtract
- “Same level” operations (mult/div/remainder or add/sub) are handled left to right

Use parentheses when you need to override precedence, clarify, or make code easier to read
- …but don’t sprinkle them in everywhere just for good measure; that’s confusing to readers

Evaluate these expressions:
- 4 - 2 + 9 / 2 — 4 - 2 + 4.5 = 2 + 4.5 = 6.5
- 4 - (2 + 9) / 2 — 4 - 11 / 2 = 4 - 5.5 = -1.5
- ((4 - 2) + (9 / 2)) — 4 - 2 + 4.5 = 2 + 4.5 = 6.5

Note: parens add no value and are potentially confusing

53



Lab Challenge #1

Write a program that…

Asks the user for their name

Ask the user for their age

Calculates their age in dog years (human years divided by seven)

Displays the result, nicely labeled, like this:

```
Yuka, you are 4.142857142857143 in dog years!
```

54

Lab Challenge #1: Bill’s Solution

```
name = input('Enter your name: ')
age = float(input('Enter your age: '))
dogYears = age / 7
print(name, ', you are ', dogYears, ' in dog years!', sep='')
```

55

Getting More Control over Output with Formatted Results

Python has a command called `format` that lets you specify…

- …the **thing** you want to format, plus…
- …a specification describing **how** you want it formatted

It creates a *string* containing the result. Use this as a *reporting* helper, not on intermediate results you may still need to use in future calculations

Here are some examples:

Code	**Result**	**Explanation**
`format(1234.567, '.2f')`	`'1234.57'`	Two digits after decimal
`format(1234.567, ',.0f')`	`'1,235'`	Commas, no digits after decimal
`format(34.567, '8.2f')`	`'   34.57'`	Eight character minimum, two digits after decimal
`format(0.63, '%')`	`'63.000000%'`	Percent notation
`format(0.63, '.0%')`	`'63%'`	Percent notation, no digits after decimal

56

Common Format Specifier Types

f Use for floating-point numbers
Example: format(1234.567, '10,.2f') → _ _ 1,234.57

% Use for floating point numbers, to show as percentages
Example: format(0.04578, '7.3%') → _ 4.578000%

d Use for integer digits
Example: format(34, '5d') → _ _ _ 34

s Use for strings (text)
Example: format('Hello', '10s') → Hello _ _ _ _ _

57

Format Specifications for the Brave

While this is not required for basic understanding, here is some additional information for those who really want to know

A format specifier is described by:

[*align*][*sign*][*width*][,][.*precision*][*type*]

…where square brackets indicate options and italics indicate “fill in the blank” options. There is more to it, but this is enough for now!

Some interesting implications:

- You can force alignment using: < > ^
- Signs are interesting options: + space

See the documentation on the Format Specification Mini Language for more information (it’s under “Common string operations”

58

Lab Challenge #2

Ask the user for their name and the amount of their purchase

Calculate tax, assuming a tax rate of 9.5%

Display an attractive report showing the results, perhaps this…

```
Hello Bill, here is your sales information:
Subtotal = $ 1,234.56
     Tax = $   117.28
             --------
   Total = $ 1,351.84
```

59

Lab Challenge #2: Bill’s Solution

```
# Constants
TAX_RATE = 0.095
OUTPUT_FMT_DOLLARS = '8,.2f'
OUTPUT_FMT_STRINGS = '>8s'

# Inputs
userName = input('Enter your name: ')
subtotal = float(input('Enter pre-tax total amount: '))

# Calculations
tax = subtotal * TAX_RATE
total = subtotal + tax

# Output
print()
print('Hello ', userName, ', here is your sales information:', sep='')
print()
print('Subtotal = $', format(subtotal,   OUTPUT_FMT_DOLLARS))
print('     Tax = $', format(tax,        OUTPUT_FMT_DOLLARS))
print('            ', format('-------',  OUTPUT_FMT_STRINGS))
print('   Total = $', format(total,      OUTPUT_FMT_DOLLARS))
print()
print('Goodbye')
```

60

The End

61