Computer Programming

Course Code: CS 501, Spring 2025

SECTION : (Friday) 2:00PM – 5:00 PM, Class Room : 614

Presented by
Dr. Rubaiyat Islam
Associate Professor,
Department of Software Engineering
Daffodil international University
Omdena Bangladesh Chapter Lead
Crypto-economist Consultant
Sifchain Finance, USA.

TODAY THEORETICAL

- course info
- python basics
- mathematical operations
- python variables and types

TODAY PRACTICAL

- Python Local environment setup
- Python cloud environment
- mathematical operations
- python variables and types

COURSE INFO

ASSESSMENT AND EVALUATION							
1	**Assessment Methods**						
	Methods	Weighting	CLO1	CLO2	CLO3	CLO4	CLO5
	Attendance	7%	0%	0%	0%	0%	0%
	Class Test	15%	2%	5%	3%	5%	0%
	Presentation	8%	0%	0%	0%	0%	8%
	Assignment	5%	1.5%	1%	1.5%	1%	0%
	Mid-Term Exam	25%	5%	20%	0%	0%	0%
	Final Exam	40%	0%	0%	15%	25%	0%
	Total	**100%**	**8.5%**	**26%**	**19.5%**	**31%**	**8%**

ASSESSMENT POLICIES

2	**Grading System**			
	Marks	Grade	Grade Point	**Remark**
	80-100%	A +	4	**Outstanding**
	75-79%	A	3.75	**Excellent**
	70-74%	A-	3.5	**Very Good**
	65-69%	B+	3.25	**Good**
	60-64%	B	3	**Satisfactory**
	55-59%	B-	2.75	**Above Average**
	50-54%	C+	2.5	**Average**
	45-49%	C	2.25	**Below Average**
	40-44%	D	2	**Pass**
	00-39%	F	0	**Fail**
3	**Make-up Procedures**			
	Improvement Exam (Students who have failed or received unsatisfactory grades (less than or equal to B) in the regular examinations and thus want to improve their grades), and Incomplete (I) Exam.			

RECITATIONS

1) Lecture review: **review** lecture material will be available in the BLC portal

2) Problem-solving: teach you **how to solve** programming problems

- useful if you don't know how to set up pseudocode from words
- we show a couple of harder questions
- walk you through how to approach solving the problem
- brainstorm code solution along with the recitation instructor
- will post solutions after

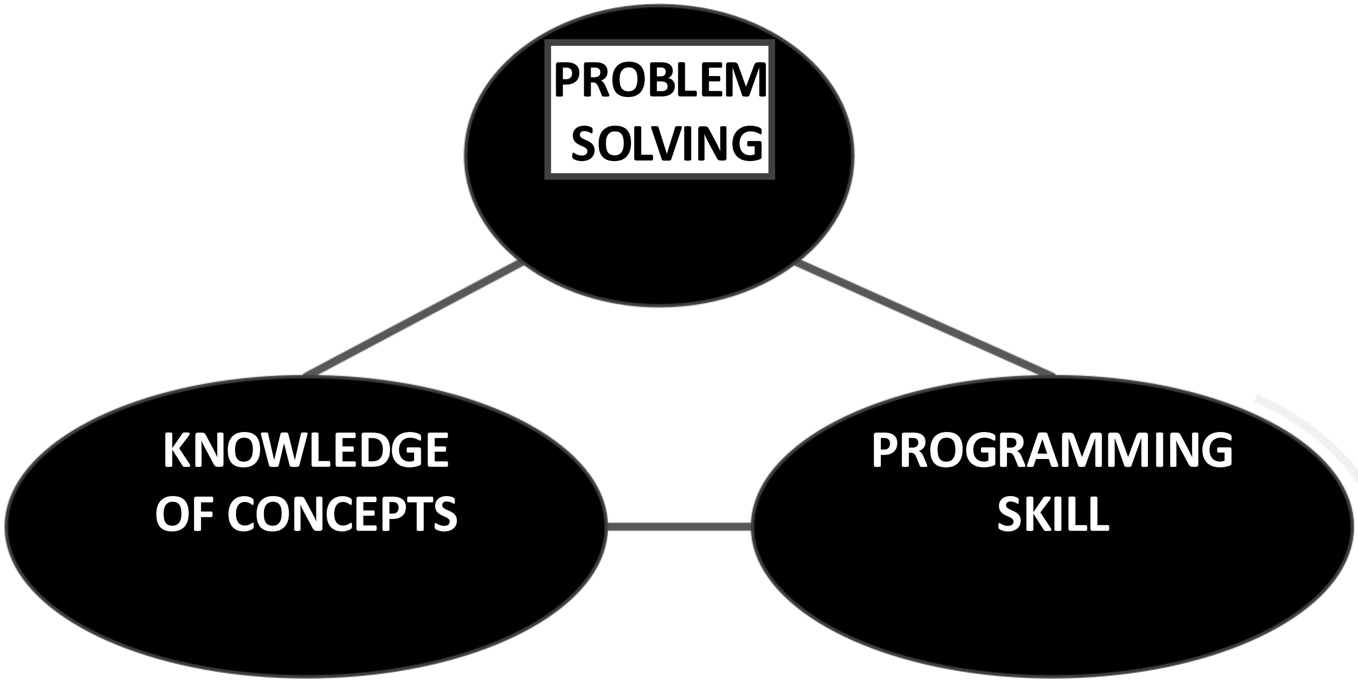
FAST PACED COURSE

- New to programming?

PRACTICE. PRACTICE? PRACTICE!

  - can’t passively absorb programming as a skill
  - download the code before the lecture and follow along
  - There will be some Reference knowledge from AI tools
  - don’t be afraid to try out Python commands!

PRACTICE



TOPICS

- represent knowledge with **data structures**
- **iteration and recursion** as computational metaphors
- **abstraction** of procedures and data types
- **organize and modularize** systems using object classes and methods
- different classes of **algorithms**, searching and sorting
- **complexity** of algorithms

WHAT DOES A COMPUTER DO

- Fundamentally:
  - performs **calculations**
    a billion calculations per second!
  - **remembers** results
    100s of gigabytes of storage!
- What kinds of calculations?
  - **built-in** to the language
  - ones that **you define** as the programmer
- computers only know what you tell them

TYPES OF KNOWLEDGE

- **declarative knowledge** is **statements of fact**.
  - someone will win a Google Cardboard before class ends
- **imperative knowledge** is a **recipe** or “how-to”.
  1) Students sign up for raffle
  2) Ana opens her IDE
  3) Ana chooses a random number between 1st and nth responder
  4) Ana finds the number in the responders sheet. Winner!

A NUMERICAL EXAMPLE

- square root of a number `x` is `y` such that `y*y = x`
- recipe for deducing square root of a number `x` (16)
  1) Start with a **guess**, `g`
  2) If `g*g` is **close enough** to `x`, stop and say `g` is the answer
  3) Otherwise make a **new guess** by averaging `g` and `x/g`
  4) Using the new guess, **repeat** process until close enough

g	g*g	x/g	(g+x/g)/2
3	9	16/3	4.17
4.17	17.36	3.837	4.0035
4.0035	16.0277	3.997	4.000002

WHAT IS A RECIPE

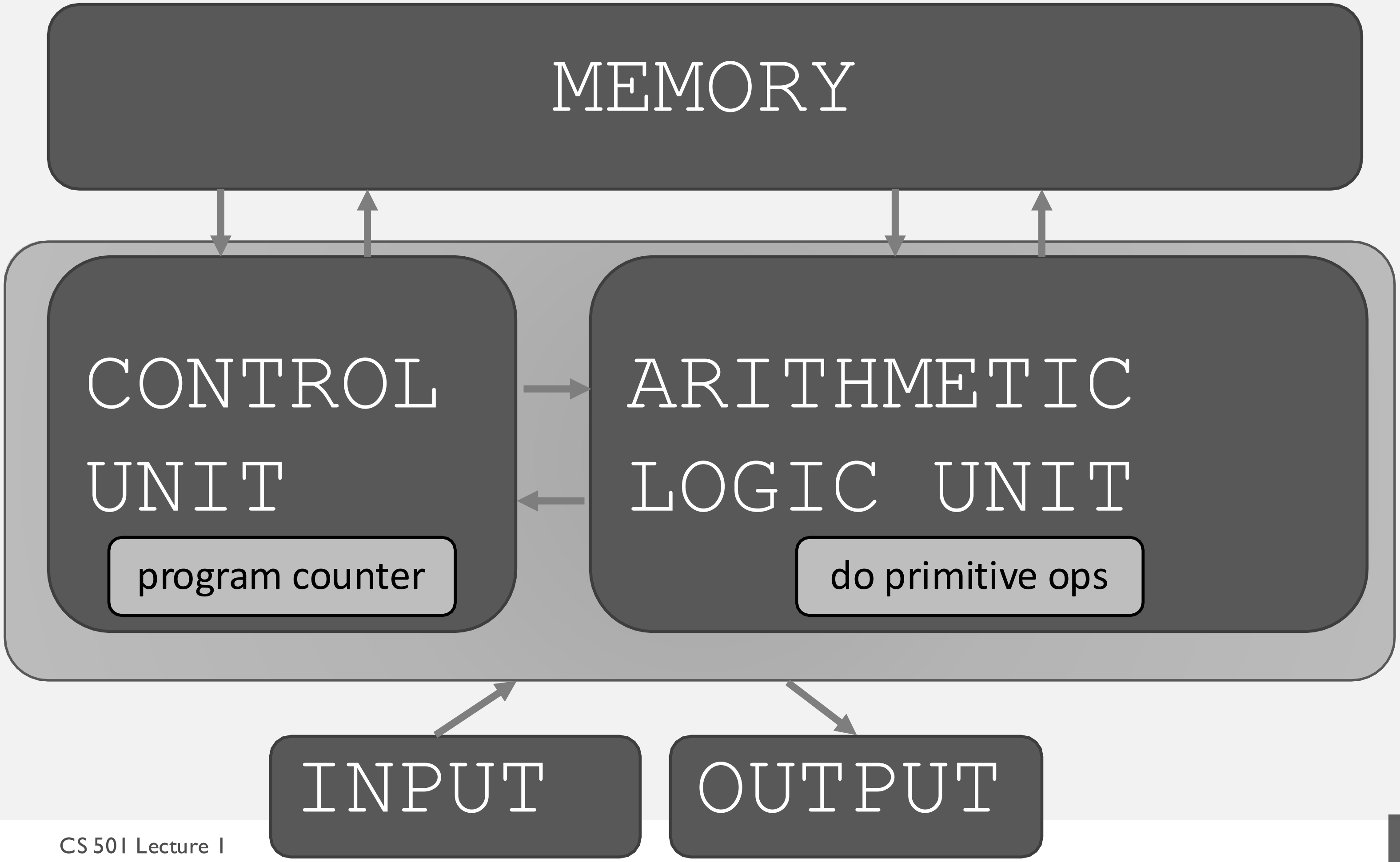
1) sequence of simple **steps**
2) **flow of control** process that specifies when each step is executed
3) a means of determining **when to stop**

1+2+3 = an **algorithm**!

COMPUTERS ARE MACHINES

- how to capture a recipe in a mechanical process
- **fixed program** computer
  - calculator
- **stored program** computer
  - machine stores and executes instructions

BASIC MACHINE ARCHITECTURE



STORED PROGRAM COMPUTER

- sequence of **instructions stored** inside computer
  - built from predefined set of primitive instructions
    1) arithmetic and logic
    2) simple tests
    3) moving data
- special program (interpreter) **executes each instruction in order**
  - use tests to change flow of control through sequence
  - stop when done

BASIC PRIMITIVES

- Turing showed that you can **compute anything** using 6 primitives
- modern programming languages have more convenient set of primitives
- can abstract methods to **create new primitives**
- anything computable in one language is computable in any other programming language

CREATING RECIPES

- a programming language provides a set of primitive **operations**
- **expressions** are complex but legal combinations of primitives in a programming language
- expressions and computations have **values** and meanings in a programming language

ASPECTS OF LANGUAGES

- **syntax**
  - English: "`cat dog boy`" → not syntactically valid
    "`cat hugs boy`" → syntactically valid
  - programming language: `"hi"5` → not syntactically valid
    `3.2*5` → syntactically valid

ASPECTS OF LANGUAGES

- **static semantics** is which syntactically valid strings have meaning
  - English: `"I are hungry"` → syntactically valid
    - but static semantic error
  - programming language: `3.2*5` → syntactically valid
    - `3+"hi"` → static semantic error

ASPECTS OF LANGUAGES

- **semantics** is the meaning associated with a syntactically correct string of symbols with no static semantic errors
  - English: can have many meanings "`Flying planes can be dangerous`"
  - programming languages: have only one meaning but may not be what programmer intended

WHERE THINGS GO WRONG

- **syntactic errors**
  - common and easily caught
- **static semantic errors**
  - some languages check for these before running program
  - can cause unpredictable behavior
- no semantic errors but **different meaning than what programmer intended**
  - program crashes, stops running
  - program runs forever
  - program gives an answer but different than expected

PYTHON PROGRAMS

- a **program** is a sequence of definitions and commands
  - definitions **evaluated**
  - commands **executed** by Python interpreter in a shell
- **commands** (statements) instruct interpreter to do something
- can be typed directly in a **shell** or stored in a **file** that is read into the shell and evaluated
  - Problem Set 0 will introduce you to these in Anaconda

OBJECTS

- programs manipulate **data objects**

- objects have a **type** that defines the kinds of things programs can do to them
  - Ana is a human so she can walk, speak English, etc.
  - Chewbacca is a wookie so he can walk, “mwaaarhrhh”, etc.
- objects are
  - scalar (cannot be subdivided)
  - non-scalar (have internal structure that can be accessed)

SCALAR OBJECTS

- `int` – represent **integers**, ex. `5`
- `float` – represent **real numbers**, ex. `3.27`
- `bool` – represent **Boolean** values `True` and `False`
- `NoneType` – **special** and has one value, `None`
- can use `type()` to see the type of an object

```
>>> type(5)
int
>>> type(3.0)
float
```

what you write into the Python shell

what shows after hitting enter

TYPE CONVERSIONS (CAST)

- can **convert object of one type to another**
- `float(3)` converts integer `3` to float `3.0`
- `int(3.9)` truncates float `3.9` to integer `3`

PRINTING TO CONSOLE

- to show output from code to a user, use `print` command

```
In [11]: 3+2
Out[11]: 5
```

"Out" tells you it's an interaction within the shell only

```
In [12]: print(3+2)
5
```

No "Out" means it is actually shown to a user, apparent when you edit/run files

EXPRESSIONS

- **combine objects and operators** to form expressions
- an expression has a **value**, which has a type
- syntax for a simple expression

```
<object> <operator> <object>
```

OPERATORS ON INTS AND FLOATS

- `i+j` → the **sum**
- `i-j` → the **difference**
- `i*j` → the **product**
- `i/j` → **division**

if both are ints, result is int
if either or both are floats, result is float

result is float

- `i%j` → the **remainder** when `i` is divided by `j`
- `i**j` → `i` to the **power** of `j`

SIMPLE OPERATIONS

- parentheses used to tell Python to do these operations first
- **operator precedence** without parentheses
  - **
  - *
  - /
  - + and – executed left to right, as appear in expression

BINDING VARIABLES AND VALUES

- equal sign is an **assignment** of a value to a variable name

variable value

```
pi = 3.14159
pi_approx = 22/7
```

- value stored in computer memory
- an assignment binds name to value
- retrieve value associated with name or variable by invoking the name, by typing `pi`

ABSTRACTING EXPRESSIONS

- why **give names** to values of expressions?
- to **reuse names** instead of values
- easier to change code later

```
pi = 3.14159
radius = 2.2
area = pi*(radius**2)
```

PROGRAMMINGVS MATH

- in programming, you do not “solve for x”

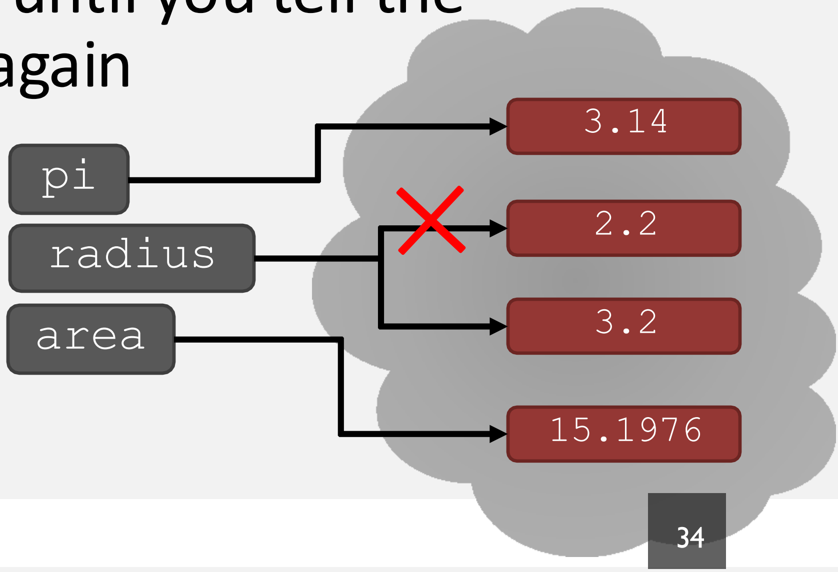
```
pi = 3.14159
radius = 2.2
# area of circle
area = pi*(radius**2)
radius = radius+1
```

an assignment
* expression on the right, evaluated to a value
* variable name on the left
* equivalent expression to `radius = radius + 1`
is `radius += 1`

CHANGING BINDINGS

- can **re-bind** variable names using new assignment statements
- previous value may still stored in memory but lost the handle for it
- value for area does not change until you tell the computer to do the calculation again

```
pi = 3.14
radius = 2.2
area = pi*(radius**2)
radius = radius+1
```



PYTHON INTRO OVERVIEW

- **Values**: `10` (integer), `3.1415` (decimal number or float), `'wellesley'` (text or string)
- **Types**: numbers and text: `int`, `float`, `str`
  `type(10)`
  `type('wellesley')`

Knowing the **type** of a **value** allows us to choose the right **operator** when creating **expressions**.

- **Operators:** + - * / % =
- **Expressions:** (they always produce a value as a result)
  `'abc' + 'def' -> 'abcdef'`

SIMPLE EXPRESSIONS: PYTHON AS CALCULATOR

Concepts in this slide: numerical values, math operators, expressions.

Input Expressions In [...]	Output Values Out [...]	
`1+2`	3	
`3*4`	12	
`3 * 4`	12	# Spaces don't matter
`3.4 * 5.67`	19.278	# Floating point (decimal) operations
`2 + 3 * 4`	14	# Precedence: * binds more tightly than +
`(2 + 3) * 4`	20	# Overriding precedence with parentheses
`11 / 4`	2.75	# Floating point (decimal) division
`11 // 4`	2	# Integer division
`11 % 4`	3	# Remainder (often called modulus)
`5 - 3.4`	1.6	
`3.25 * 4`	13.0	
`11.0 // 2`	5.0	# output is float if at least one input is float
`5 // 2.25`	2.0	
`5 % 2.25`	0.5	

BUILT-IN FUNCTIONS:

Built-in function	Result
`max`	Returns the largest item in an iterable (an iterable is an object we can loop over, like a list of numbers. We will learn about them soon!)
`min`	Returns the smallest item in an iterable
`id`	Returns memory address of a value
`type`	Returns the type of a value
`len`	Returns the length of a sequence value (strings are an example)
`str`	Converts and returns the input as a string
`int`	Converts and returns the input as an integer number
`float`	Converts and returns the input as a floating point number
`round`	Rounds a number to nearest integer or decimal point
`print`	Prints a specified message on the screen/output device,, and returns the `None` value.
`input`	Asks user for input, converts input to a string, returns the string

BUILT-IN FUNCTIONS: MAX AND MIN

Concepts in this slide:
built-in functions,
arguments,
function calls.

Python has many built-in functions that we can use. Built-in functions and user-defined variable and function names names are highlighted with different colors in both Thonny and Jupyter Notebooks.

In [...] **Out [...]**

```
min(7, 3)                      3
max(7, 3)                      7
min(7, 3, 2, 8.19)             2 # can take any num. of arguments
max(7, 3, 2, 8.19)             8.19
smallest = min(-5, 2)           # smallest gets -5
largest = max(-3.4, -10)        # largest gets -3.4
max(smallest, largest, -1)     -1
```

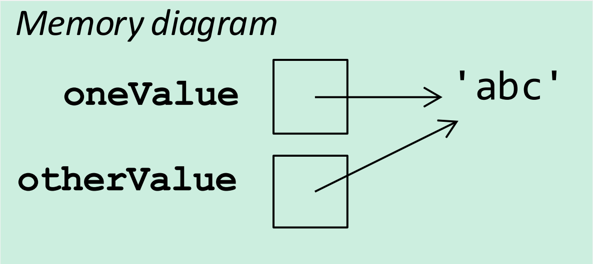
The inputs to a function are called its **arguments** and the function is said to be **called** on its arguments. In Python, the arguments in a function call are delimited by parentheses and separated by commas.

UNDERSTANDING VARIABLE AND FUNCTION NAMES

Concepts in this slide: Values can have multiple names. Functions are also values.

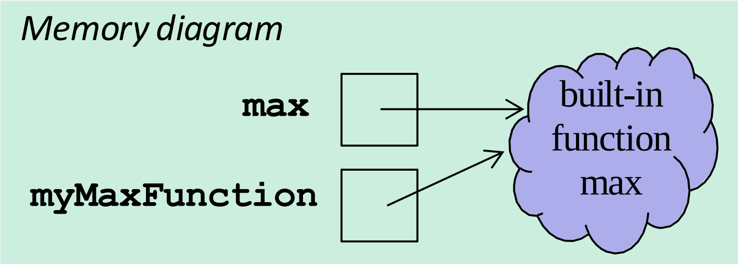
One value can have multiple names. These names refer to the same value in the computer memory. See the examples below for variables and functions.

```
>>> oneValue = 'abc'
>>> otherValue = oneValue
>>> oneValue
'abc'
>>> otherValue
'abc'
```



Functions are values. Just like numbers & strings

```
>>> max
<built-in function max>
>>> myMaxFunction = max
>>> max(10,100)
100
>>> myMaxFunction(10,100)
100
```



BUILT-IN FUNCTIONS: `ID`

Concepts in this slide:
Values can have multiple names. Functions are also values.

```
>>> id(oneValue)
4526040688
>>> id(otherValue)
4526040688
```

Built-in function id:
This function displays the memory address where a value is stored.

Different names can refer to the same value in memory.

```
>>> id(max)
4525077120
>>> id(myMaxFunction)
4525077120
```

BUILT-IN FUNCTIONS: `TYPE`

Concepts in this slide:
types,
the function **`type`**.

Each Python value has a type. It can be queried with the built-in **`type`** function.

Types are special kinds of values that display as **`<class '`*`typeName`*`'>`** Knowing the type of a value is important for reasoning about expressions containing the value.

In [...]	Out [...]
`type(123)`	`int`
`type(3.141)`	`float`
`type(4 + 5.0)`	`float`
`type('CS111')`	`str`
`type('111')`	`str`
`type(11/4)`	`float`
`type(11//4)`	`int`
`type(11%4)`	`int`
`type(11.0%4)`	`float`
`type(max(7, 3.4))`	`int`
`x = min(7, 3.4)`	`# x gets 3.4`
`type(x)`	`float`
`type('Hi,' + 'you!')`	`str`
`type(type(111))`	`type # Special type for types!`

Jupyter notebooks display these type names. Thonny actually displays **`<class 'int'>`** , **`<class 'float'>`** , etc., but we'll often abbreviate these using the Jupyter notebook types **`int`**, **`float`**, etc.

USING `TYPE` WITH DIFFERENT VALUES

Concepts in this slide: Every value in Python has a type, which can be queried with **`type`**.

Below are some examples of using **`type`** in Thonny, with different values:

```
>>> type(10)
<class 'int'>

>>> type('abc')
<class 'str'>

>>> type(10/3)
<class 'float'>

>>> type(max)
<class 'builtin_function_or_method'>

>>> type(len)
<class 'builtin_function_or_method'>

>>> type(True)
<class 'bool'>

>>> type([1,2,3])
<class 'list'>

>>> type((10,5))
<class 'tuple'>
```

Functions are values with this type

Other types we will learn about later in the semester

BUILT-IN FUNCTIONS: LEN

Concepts in this slide:
length of a string,
the function `len`,
TypeError

When applied to a **string**, the built-in `len` function returns the number of characters in the string.

`len` raises a `TypeError` if used on values (like numbers) that are not sequences. (We'll learn about sequences later in the course.)

In [...]	Out [...]
`len('CS111')`	5
`len('CS111 rocks!')`	12
`len('com' + 'puter')`	8
`course = 'computer programming'`	
`len(course)`	20
`len(111)`	TypeError
`len('111')`	3
`len(3.141)`	TypeError
`len('3.141')`	5

BUILT-IN FUNCTIONS: STR

Concepts in this slide:
the `str` function

The `str` built-in function returns a string representation of its argument.

It is used to create string values from `int`s and `float`s (and other types of values we will meet later) to use in expressions with other string values.

In […]	Out […]
`str('CS111')`	`'CS111'`
`str(17)`	`'17'`
`str(4.0)`	`'4.0'`
`'CS' + 111`	`TypeError`
`'CS' + str(111)`	`'CS111'`
`len(str(111))`	`3`
`len(str(min(111, 42)))`	`2`

BUILT-IN FUNCTIONS: INT

Concepts in this slide:
int function,
TypeError,
ValueError.

- When given a string that's a sequence of digits, optionally preceded by +/-, **int** returns the corresponding integer. On any other string it raises a **ValueError** (correct type, but wrong value of that type).
- When given a float, **int** return the integer the results by truncating it toward zero.
- When given an integer, **int** returns that integer.

In [...]	Out [...]	
`int('42')`	`42`	
`int('-273')`	`-273`	
`123 + '42'`	`TypeError`	
`123 + int('42')`	`165`	
`int('3.141')`	`ValueError`	`# strings are not sequence`
`int('five')`	`ValueError`	`# of chars denoting integer`
`int(3.141)`	`3`	
`int(98.6)`	`98`	`# Truncate floats toward 0`
`int(-2.978)`	`-2`	
`int(42)`	`42`	
`Int(-273)`	`-273`	

BUILT-IN FUNCTIONS: `FLOAT`

Concepts in this slide:
`float` function,
ValueError

- When given a string that's a sequence of digits, optionally preceded by +/-, and optionally including one decimal point, **`float`** returns the corresponding floating point number. On any other string it raises a **`ValueError`**.
- When given an integer, **`float`** converts it to floating point number.
- When given a floating point number, **`float`** returns that number.

In [...]	Out [...]
`float('3.141')`	`3.141`
`float('-273.15')`	`-273.15`
`float('3')`	`3.0`
`float('3.1.4')`	`ValueError`
`float('pi')`	`ValueError`
`float(42)`	`42.0`
`float(98.6)`	`98.6`

ODDITIES OF FLOATING POINT NUMBERS

Concepts in this slide:
floating point numbers are only approximations, so don't always behave exactly like math

In computer languages, floating point numbers (numbers with decimal points) don't always behave like you might expect from mathematics. This is a consequence of their fixed-sized internal representations, which permit only approximations in many cases.

In [...]	Out [...]
`2.1 - 2.0`	`0.10000000000000009`
`2.2 - 2.0`	`0.20000000000000018`
`2.3 - 2.0`	`0.2999999999999998`
`1.3 - 1.0`	`0.30000000000000004`
`100.3 - 100.0`	`0.29999999999999716`
`10.0/3.0`	`3.3333333333333335`
`1.414*(3.14159/1.414)`	`3.1415900000000003`

BUILT-IN FUNCTIONS: ROUND

Concepts in this slide: the `round` function, called with varying number of arguments.

- When given **one** numeric argument, `round` returns the **integer** it's closest to.
- When given **two** arguments (a numeric argument and an integer number of decimal places), `round` returns **floating point** result of rounding the first argument to the number of places specified by the second.
- In other cases, `round` raises a `TypeError`

In [...]	Out [...]	
`round(3.14156)`	3	
`round(98.6)`	99	
`round(-98.6)`	-99	
`round(3.5)`	4	# always rounds up for 0.5
`round(4.5)`	5	
`round(2.718, 2)`	2.72	
`round(2.718, 1)`	2.7	
`round(2.718, 0)`	3.0	
`round(1.3 - 1.0, 1)`	0.3	# Compare to previous slide
`round(2.3 - 2.0, 1)`	0.3	

BUILT-IN FUNCTIONS: `PRINT`

Concepts in this slide:
`print` function

`print` displays a character-based representation of its argument(s) on the screen and **returns** a special `None` value (not displayed).

Input statements In [...]	Characters displayed in console (*not* the output value of the expression!)
`print(7)`	`7`
`print('CS111')`	`CS111`
`print(len(str('CS111')) * min(17,3))`	`15`
`college = 'Wellesley'`	
`print('I go to ' + college)`	`I go to Wellesley`
`dollars = 10`	
`print('The movie costs $' + str(dollars) + '.')`	`The movie costs $10.`

THE NEWLINE CHARACTER '\N'

Concepts in this slide:
The `'\n'` newline character.

`'\n'` is a single special **newline character**. **Printing it causes the console to shift to the next line.**

In [...]

```
print('one\ntwo\nthree')
```

Console

```
one
two
three
```

PRINT WITH MULTIPLE ARGUMENTS

Concepts in this slide:
print can take more than one argument

When **print** is given more than one argument, it prints all arguments, separated by one space by default. This is helpful for avoiding concatenating the parts of the printed string using **+** and using **str** to convert nonstrings to strings.

In [...]

Console

```
print(6,'*',7,'=',6*7)
```

```
6 * 7 = 42
```

```
# print with one argument is much
# more complicated in this example!
print(str(6)+' * '+str(7)+' = '+str(6*7))
```

```
6 * 7 = 42
```

PRINT WITH THE SEP KEYWORD ARGUMENT

Concepts in this slide:
The optional `sep` keyword argument overrides the default space between values

`print` can take an optional so-called *keyword argument* of the form `sep=`*stringValue* that uses *stringValue* to replace the default space string between multiple values.

In [...]

```
print(6,'*',7,'=',6*7)

# replace space by $
print(6,'*',7,'=',6*7,sep='$')

# replace space by two spaces
print(6,'*',7,'=',6*7,sep='  ')

# replace space by zero spaces
print(6, '*',7,'=',6*7,sep='')

# replace space by newline
print(6,'*',7,'=',6*7,sep='\n')
```

Console

```
6 * 7 = 42

6$*$7$=$42

6  *  7  =  42

6*7=42

6
*
7
=
42
```

PRINT RETURNS NONE!

Concepts in this slide:
The optional `sep` keyword argument overrides the default space between values

In addition to printing characters in the console, `print` also **returns** the special value `None`. Confusingly, but Thonny and Jupyter notebooks do not explicitly display this `None`. value, but there are still ways to see that it's really there.

```
 In [1]: str(print('Hi!'))
         Hi! # printed by print
Out [1]: 'None' # string value returned by str

 In [2]: print(print(6*7))
         42 # printed by 2nd print
         None # printed by 1st print
         # No Out [2] shown when result is None

 In [3]: type(print(print('CS'),print(111)))
         CS # printed by 2nd print
         111 # printed by 3rd print
         None None # printed by 1st print
Out [3]: NoneType # The type of None is NoneType
```

MORE PRINT EXAMPLES

Concepts in this slide:
The '\n' newline character ; **print** returns the **None** value, which is normally hidden.

```
In [8]: print('one\ntwo\nthree')     # '\n' is a single special
one                                  # newline character.
two                                  # Printing it causes the
three                                # display to shift to the
                                     # next line.

In [9]: print('one', 'two', 'three', sep='\n')
one                                  # Like previous example,
two                                  # but use sep keyword arg
three                                # for newlines

In [10]: str(print(print('CS'), print(111)))
CS  # printed by 2nd print.
111 # printed by 3rd print.
None None  # printed by 1st print; shows that print returns None
Out[10]: 'None'  # Output of str; shows that print returns None
```

BUILT-IN FUNCTIONS: INPUT

Concepts in this slide:
The `input` function;
converting from string
returned by `input`.

`input` displays its single argument as a prompt on the screen and waits for the user to input text, followed by Enter/Return. It returns the entered value as a **string**.

```
In [1]: input('Enter your name: ')
Enter your name: Olivia Rodrigo
```

Magenta text is entered by user.

Brown text is prompt.

```
Out [1]: 'Olivia Rodrigo'
```

BUILT-IN FUNCTIONS: INPUT

Concepts in this slide:
The `input` function;
converting from string
returned by `input`.

```
In [2]: age = input('Enter your age: ')
Enter your age:20
```

No output from assignment.

```
In [3]: age
Out [3]: '20'
```

Value returned by `input` is always a **string**.
Convert it to a numerical type when needed.

```
In [4]: age + 4
TypeError
```

Tried to add a string and a float.

BUILT-IN FUNCTIONS: `INPUT`

Concepts in this slide:
The `input` function;
converting from string returned by `input`.

```
In [5]: age = float(input('Enter your age: '))
Enter your age: 18

In [6]: age + 4

Out [6]: 22.0
```

Example of nested function calls.

`age` contains `float('18')`, which is `18.0` and `18.0 + 4` is `22.0`

COMPLEX EXPRESSION EVALUATION

Concepts in this slide:
complex expressions ;
subexpressions;
expression evaluation

An **expression** is a programming language phrase that denotes a value. Smaller **sub-expressions** can be combined to form arbitrarily large expressions.

Complex expressions are evaluated from "inside out", first finding the value of smaller expressions, and then combining those to yield the values of larger expressions. See how the expression below evaluates to `'35'`:

```
str((3 + 4) * len('C' + 'S' + str(max(110, 111))))

7    'CS'    111
'111' # str(111)
'CS111' # 'CS' + '111'
5  # len('CS111')
35  # 7 * 5
'35'  # str(35)
```

EXPRESSIONS vs. Statements

- They always **produce a value**:

- `10`
- `10 * 20 - 100/25`
- `max(10, 20)`
- `int("100") + 200`
- `fav`
- `fav + 3`
- `"pie" + " in the sky"`

- Expressions are composed of any combination of values, variables operations, and function calls.

They **perform an action** (that can be visible, invisible, or both):

```
print(10)
age = 19
teleport(0, 150)
```

Statements may contain expressions, which are evaluated **before** the action is performed.

```
print('She is ' + str(age) + ' years old.')
```

Some statements return a `None` value that is not normally displayed in Thonny or Jupyter notebooks.

EXPRESSIONS, STATEMENTS, AND CONSOLE PRINTING IN JUPYTER

Concepts in this slide:
Jupyter displays `Out[]` for expressions, but not statemen
Non-`Out[]` chars come from p

```
In [1]: max(10,20)
Out[1]: 20

In [2]: 10 + 20
Out[2]: 30

In [3]: message = "Welcome to CS 111"

In [4]: message
Out[4]: 'Welcome to CS 111'

In [5]: print(message)
Welcome to CS 111

In [6]: print(max(10,20))
20

In [7]: print(10 + 20)
30
```

Notice the `Out[]` field for the result when the input is an expression.

EXPRESSIONS, STATEMENTS, AND CONSOLE PRINTING IN JUPYTER

Concepts in this slide:
Jupyter displays `Out[]` for expressions, but not statements.
Non-`Out[]` chars come from `print`

```
In [1]: max(10,20)
Out[1]: 20

In [2]: 10 + 20
Out[2]: 30

In [3]: message = "Welcome to CS 111"

In [4]: message
Out[4]: 'Welcome to CS 111'

In [5]: print(message)
Welcome to CS 111

In [6]: print(max(10,20))
20

In [7]: print(10 + 20)
30
```

An assignment is a statement without any outputs

The `print` function returns a `None` value that is not displayed as an output in Jupyter.
Any function or method call that returns `None` is treated as a statement in Python.

EXPRESSIONS, STATEMENTS, AND CONSOLE PRINTING IN JUPYTER

Concepts in this slide:
Jupyter displays `Out[]` for expressions, but not statements. Non-`Out[]` chars come from `print`

```
In [1]: max(10,20)
Out[1]: 20

In [2]: 10 + 20
Out[2]: 30

In [3]: message = "Welcome to CS 111"

In [4]: message
Out[4]: 'Welcome to CS 111'

In [5]: print(message)
Welcome to CS 111

In [6]: print(max(10,20))
20

In [7]: print(10 + 20)
30
```

These are characters displayed by `print` in the "console", which is interleaved with `In[]`/`Out[]`

EXPRESSIONS, STATEMENTS, AND CONSOLE PRINTING IN VSCODE

Concepts in this slide:
Thonny displays expressions, but not statements. Expressions are distinguished from printed output by text size and indentation.

```
>>> max(10, 20)
20

>>> 10 + 20
30

>>> message = "Welcome to CS 111"
>>> message
'Welcome to CS 111'

>>> print(message)

  Welcome to CS 111

>>> print(max(10, 20))

  20

>>> print(10 + 20)

  30
```

Notice no `Out[]` field for the result when the input is an expression for Thonny. Text is bigger and has no indent!

EXPRESSIONS, STATEMENTS, AND CONSOLE PRINTING IN VSCODE

Concepts in this slide:
Thonny displays expressions, but not statements. Expressions are distinguished from printed output by text size and indentation.

```
>>> max(10, 20)
20

>>> 10 + 20
30

>>> message = "Welcome to CS 111"
>>> message
'Welcome to CS 111'

>>> print(message)

   Welcome to CS 111

>>> print(max(10, 20))

   20

>>> print(10 + 20)

   30
```

An assignment is a statement without any outputs

The **print** function returns a **None** value that is not displayed as an output in Thonny.
The text is displayed as smaller and indented!

EXPRESSIONS, STATEMENTS, AND CONSOLE PRINTING IN VSCODE

Concepts in this slide:
Thonny displays expressions, but not statements. Expressions are distinguished from printed output by text size and indentation.

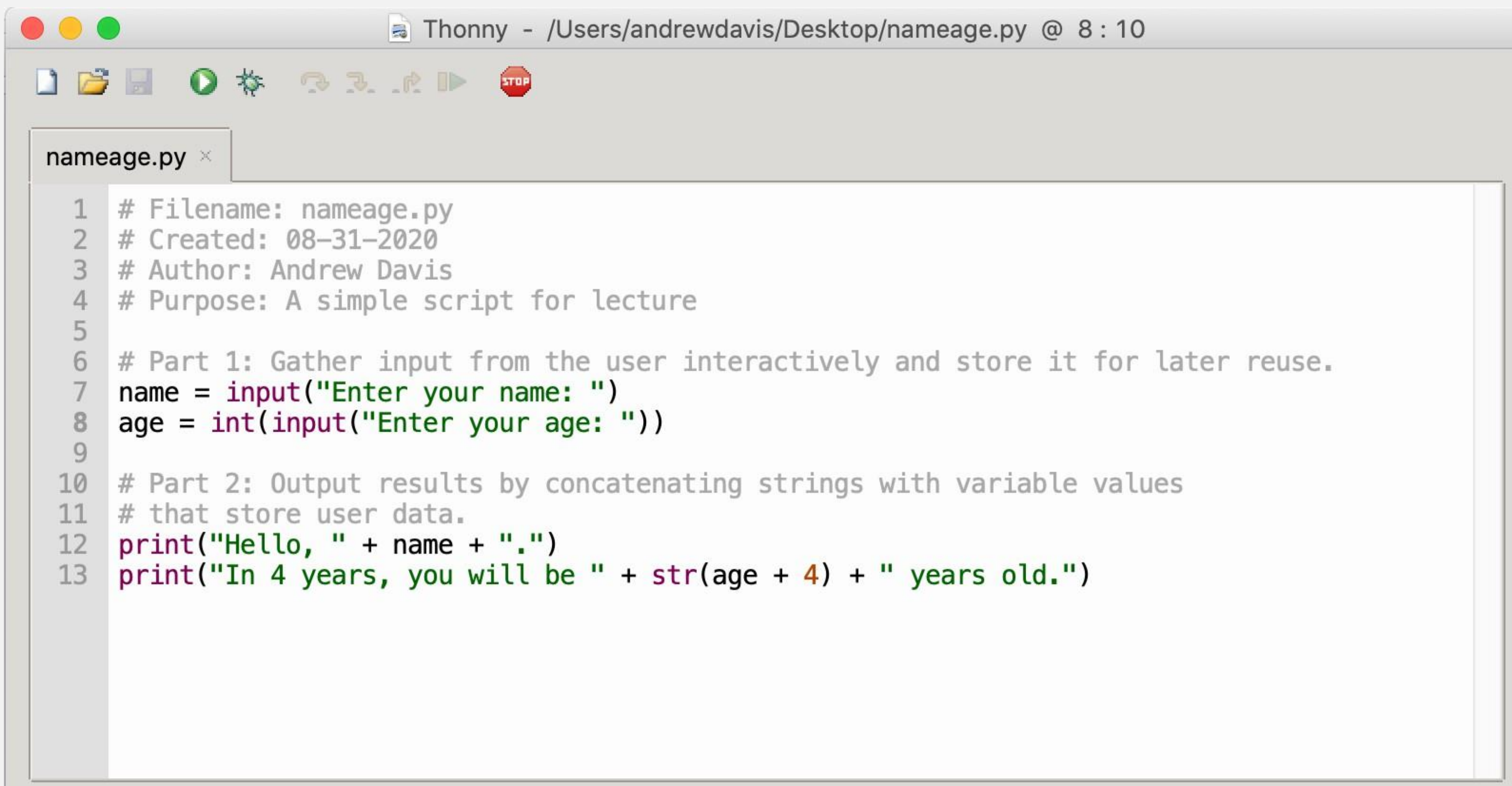
```
>>> max(10, 20)
20

>>> 10 + 20
30

>>> message = "Welcome to CS 111"
>>> message
'Welcome to CS 111'

>>> print(message)

   Welcome to CS 111

>>> print(max(10, 20))

   20

>>> print(10 + 20)

   30
```

These are characters displayed by `print` in the "console", which is interleaved with expressions

PUTTING PYTHON CODE IN A .PY FILE

Concepts in this slide: Editor pane. .py Python program file, running a program.

Rather than interactively entering code into the **Python Shell**, we can enter it in the **Editor Pane**, where we can edit it and save it away as a file with the **`.py`** extension (a Python program). Here is a **`nameage.py`** program. Lines beginning with **#** are comments We run the program by pressing the triangular "run"/play button.



Concepts in this slide:
Error types,
Error messages.

ERROR MESSAGES IN PYTHON

Type Errors

`'111' + 5` **TypeError**: cannot concatenate 'str' and 'int' values

`len(111)` **TypeError**: object of type 'int' has no len()

Value Errors

`int('3.142')` **ValueError**: invalid literal for int() with base 10: '3.142'

`float('pi')` **ValueError**: could not convert string to float: pi

Name Errors

`CS + '111'` **NameError**: name 'CS' is not defined

Syntax Errors

A syntax error indicates a phrase is not well formed according to the rules of the Python language. E.g. a number can't be added to a statement, and variable names can't begin with digits.

```
1 + (ans=42)
1 + (ans=42)
        ^
SyntaxError: invalid syntax
```

```
2ndValue = 25
2ndValue = 25
      ^
SyntaxError: invalid syntax
```

Thank You