

# Introduction to the Python language



**CS111 Computer Programming**

Department of Computer Science  
Wellesley College

# 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)

`len('abc') * 'abc' + 'def'`

- **Built-in functions:** `max`, `min`, `len`, `int`, `float`,  
`str`, `round`, `print`, `input`

# 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
5 - 3.4	1.6	} # output is float if at least one input is float
3.25 * 4	13.0	
11.0 // 2	5.0	
5 // 2.25	2.0	
5 % 2.25	0.5	

# Strings and concatenation

**Concepts in this slide:**

string values,  
string operators,  
TypeError

A string is just a sequence of characters that we write between a pair of double quotes or a pair of single quotes. Strings are usually displayed with single quotes. The same string value is created regardless of which quotes are used.

In [...]

Out [...]

```
"CS111"
```

```
'CS111' # Double quotes
```

```
'rocks!'
```

```
'rocks!' # Single quotes
```

```
'You say "Hi!"'
```

```
'You say "Hi!"' # Characters in a string
```

```
"No, I didn't"
```

```
"No, I didn't" # can include spaces,  
# punctuation, quotes
```

```
"CS111 " + 'rocks!' 'CS111 rocks!' # String concatenation
```

```
'123' + '4' '1234' # Strings and numbers
```

```
123 + 4 127 # are very different!
```

```
'123' + 4 TypeError # Can't concatenate strings & num.
```

```
'123' * 4 '123123123123' # Repeated concatenation
```

```
'123' * '4' TypeError
```

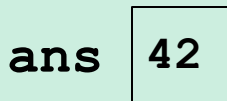


# Memory Diagram Model: Variable as a Box

**Concepts in this slide:**  
variables,  
assignment statement,  
memory diagram model,  
NameError

```
# create var box  
# via assignment  
ans = 42
```

Memory diagram



- A variable is a way to remember a value for later in the computer's memory.
- A variable is created by an **assignment statement**, whose form is **`varName = expression`**

This is executed in two steps:

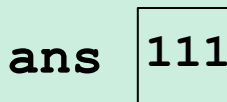
1. Evaluate **`expression`** to its value **`val`**
2. If there is no variable box already labeled with **`varName`** create a new box labeled with **`varName`** and store **`val`** in it; otherwise, change the contents of the existing box labeled **`varName`** to **`val`**.

- When **`varName`** is used in an expression, it evaluates to the current value in the box labeled **`varName`**; if there is no such box, a **NameError** occurs.
- Python variable names can contain letters, numbers, and underscores, but not other characters; they cannot begin with numbers.

```
# lookup var value  
# in expressions  
2*ans+27 # val is 111
```

```
# can lookup and  
# reassign the same var  
ans = 2*ans+27
```

Memory diagram



# Variable Examples

**Concepts in this slide:**  
variables,  
assignment statement,  
memory diagram model

A variable names a value that we want to use later in a program. In the **memory diagram model**, an assignment statement `var = exp` stores the value of `exp` in a box labeled by the variable name. Later assignments can change the value in a variable box.

**Note:** The symbol `=` is pronounced “gets” not “equals”!

```
In [...]      Memory Diagram      Out [...]
fav = 17      fav 17 12 4 # assignment stmt makes box, has no output
fav                                                  17 # returns current contents of fav box
fav + fav                                          34 # contents of fav is unchanged
lucky = 8     lucky 8 # makes new box, has no output
fav + lucky                                       25
aSum = fav + lucky  aSum 25 # makes new box, has no output
aSum * aSum                                           625
fav = 12                                             # change contents of fav box to 12
fav = fav - lucky                                   # change contents of fav box to 4
name = 'CS111'  name 'CS111' # makes new box that contains a string
name * fav                                          'CS111CS111CS111CS111'
```

# 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 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.

# Built-in functions: `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 [...]
<code>type(123)</code>	<code>int</code>
<code>type(3.141)</code>	<code>float</code>
<code>type(4 + 5.0)</code>	<code>float</code>
<code>type('CS111')</code>	<code>str</code>
<code>type('111')</code>	<code>str</code>
<code>type(11/4)</code>	<code>float</code>
<code>type(11//4)</code>	<code>int</code>
<code>type(11%4)</code>	<code>int</code>
<code>type(11.0%4)</code>	<code>float</code>
<code>type(max(7, 3.4))</code>	<code>int</code>
<code>x = min(7, 3.4)</code>	<code># x gets 3.4</code>
<code>type(x)</code>	<code>Float</code>
<code>type('Hi,' + 'you!')</code>	<code>str</code>
<code>type(type(111))</code>	<code>type # Special type for types!</code>



# Built-in functions: `len`

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

`len` raises a **`TypeError`** if used with non-string values.

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

## Built-in functions: `str`

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

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

In [...]	Out [...]
<code>str('CS111')</code>	<code>'CS111'</code>
<code>str(17)</code>	<code>'17'</code>
<code>str(4.0)</code>	<code>'4.0'</code>
<code>'CS' + 111</code>	<code>TypeError</code>
<code>'CS' + str(111)</code>	<code>'CS111'</code>
<code>len(str(111))</code>	<code>3</code>
<code>len(str(min(111, 42)))</code>	<code>2</code>

# Built-in functions: `int`

- 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 [...]
<code>int('42')</code>	42
<code>int('-273')</code>	-273
<code>123 + '42'</code>	<code>TypeError</code>
<code>123 + int('42')</code>	165
<code>int('3.141')</code>	<code>ValueError</code>
<code>int('five')</code>	<code>ValueError</code>
<code>int(3.141)</code>	3
<code>int(98.6)</code>	98
<code>int(-2.978)</code>	-2
<code>int(42)</code>	42
<code>Int(-273)</code>	-273

# strings are not sequence  
# of chars denoting integer

# Truncate floats toward 0

# Built-in functions: `float`

- 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 [...]
<code>float('3.141')</code>	<code>3.141</code>
<code>float('-273.15')</code>	<code>-273.15</code>
<code>float('3')</code>	<code>3.0</code>
<code>float('3.1.4')</code>	<code>ValueError</code>
<code>float('pi')</code>	<code>ValueError</code>
<code>float(42)</code>	<code>42.0</code>
<code>float(98.6)</code>	<code>98.6</code>

# 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 [...]**

2.1 - 2.0

2.2 - 2.0

2.3 - 2.0

1.3 - 1.0

100.3 - 100.0

10.0/3.0

1.414\*(3.14159/1.414)

**Out [...]**

0.100000000000000009

0.200000000000000018

0.29999999999999998

0.300000000000000004

0.2999999999999999716

3.33333333333333335

3.14159000000000003

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

# Built-in functions: **round**

- 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 [...]	
<code>round(3.14156)</code>	<code>3</code>	
<code>round(98.6)</code>	<code>99</code>	
<code>round(-98.6)</code>	<code>-99</code>	
<code>round(3.5)</code>	<code>4</code>	
<code>round(4.5)</code>	<code>5</code>	
<code>round(2.718, 2)</code>	<code>2.72</code>	
<code>round(2.718, 1)</code>	<code>2.7</code>	
<code>round(2.718, 0)</code>	<code>3.0</code>	
<code>round(1.3 - 1.0, 1)</code>	<code>0.3</code>	} # Compare to previous slide
<code>round(2.3 - 2.0, 1)</code>	<code>0.3</code>	

# 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

Input statements	Characters displayed in console (*not* the output value of the expression!)
<code>In [...]</code>	
<code>print(7)</code>	7
<code>print('CS111')</code>	CS111
<code>print('CS' + 111)</code>	<code>TypeError</code>
<code>print(len(str('CS111')) * min(17,3))</code>	15
<code>college = 'Wellesley'</code>	
<code>print('I go to ' + college)</code>	I go to Wellesley
<code>dollars = 10</code>	
<code>print('The movie costs \$' + str(dollars) + '.')</code>	The movie costs \$10.
<code>print(1, '+', 2, '=', 1+2)</code>	1 + 2 = 3
<code>print(1, '+', 2, '=', 1+2, sep='')</code>	1+2=3

The last two examples show that when `print` is given more than one argument, it prints all arguments, separated by one space by default. The default can be overridden by a so-called keyword argument named `sep`.

# 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
```

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

Brown text is prompt.

Magenta text is entered by user.

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

```
Enter your age: 18
```

```
In [3]: age
```

```
Out [3]: '18'
```

No output from assignment.

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.

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

```
Enter a number: 18
```

Example of nested function calls.

```
In [6]: age + 4
```

```
Out [6]: 22.0
```

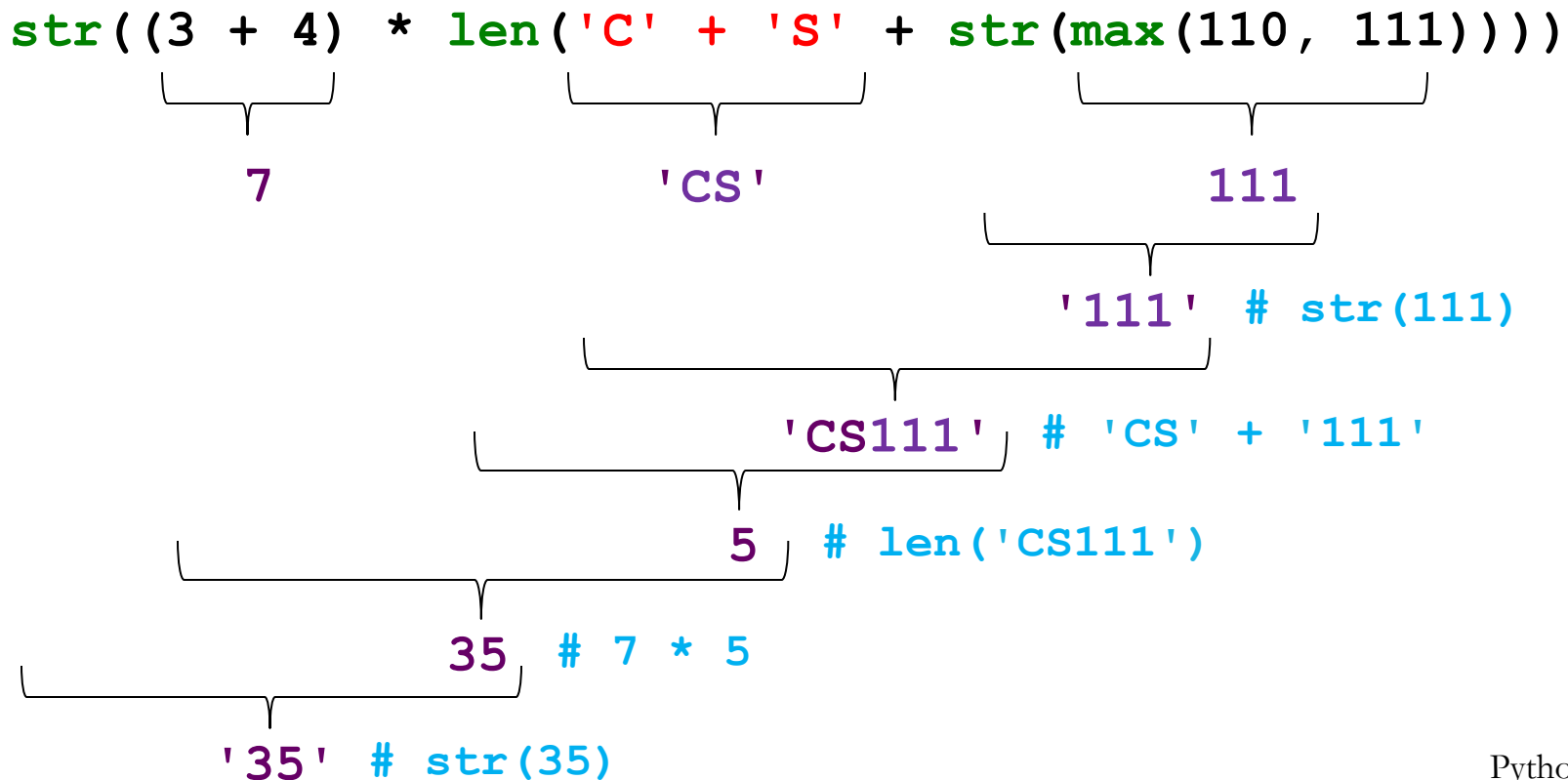
`age` contains `float('18')`, which is `18.0`  
and `18.0 + 4` is `22.0`

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

# Complex Expression Evaluation

An **expression** is a programming language phrase that denotes a value. Smaller subexpressions 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'**:



# 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
paper = Canvas(400, 550,
               'yellow')
paper.add(head)
```

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.

# Expressions, statements, and console printing in Jupyter

```
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
```

## Concepts in this slide:

Jupyter displays `Out[]` for expressions, but not statements.

Non-`Out[]` chars come from `print`

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

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.

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

# Expressions, statements, and console printing in Thonny

```
>>> 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
```

## Concepts in this slide:

Thonny displays expressions, but not statements. Expressions are distinguished from printed output by text size and indentation.

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

An assignment is a statement without any outputs

The **print** function returns a **None** value that is not displayed as an output in Thonny.

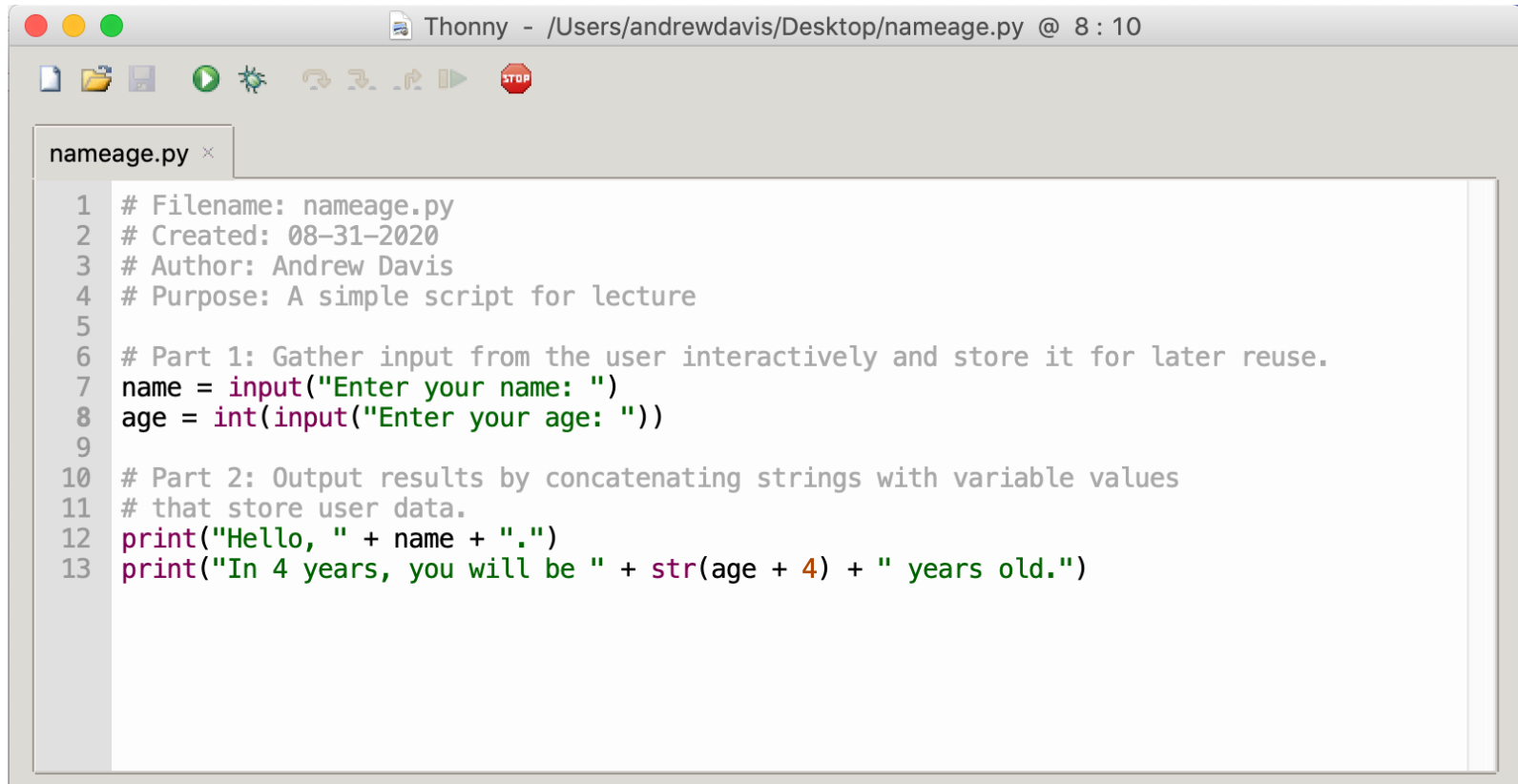
The text is displayed as smaller and indented!

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” button.

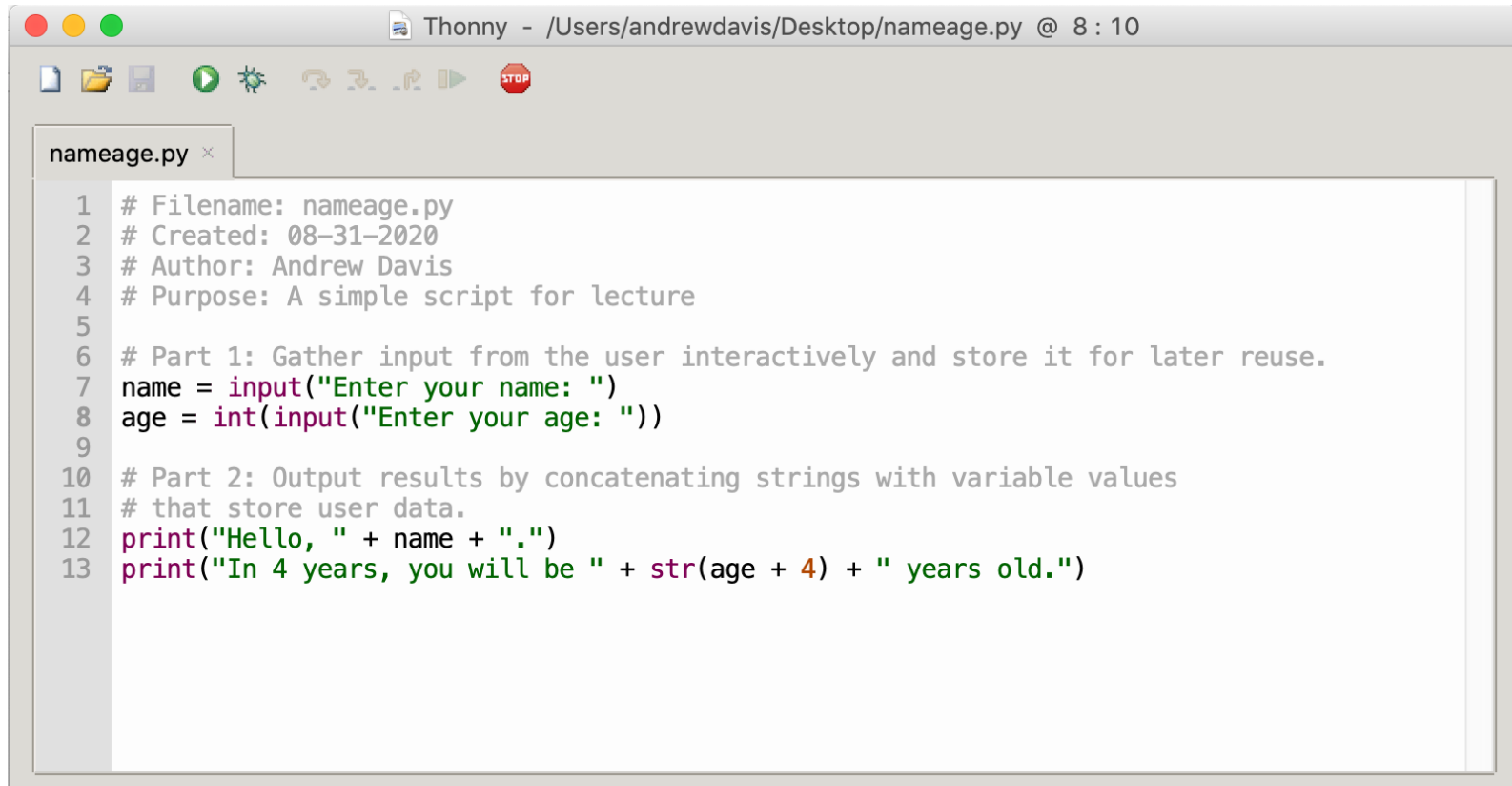


```
Thonny - /Users/andrewdavis/Desktop/nameage.py @ 8 : 10

nameage.py x
1 # Filename: nameage.py
2 # Created: 08-31-2020
3 # Author: Andrew Davis
4 # Purpose: A simple script for lecture
5
6 # Part 1: Gather input from the user interactively and store it for later reuse.
7 name = input("Enter your name: ")
8 age = int(input("Enter your age: "))
9
10 # Part 2: Output results by concatenating strings with variable values
11 # that store user data.
12 print("Hello, " + name + ".")
13 print("In 4 years, you will be " + str(age + 4) + " years old.")
```

# Code Styling Advice

**Concepts in this slide:**  
the 80-character limit,  
coding advice.



```
Thonny - /Users/andrewdavis/Desktop/nameage.py @ 8:10

nameage.py x
1 # Filename: nameage.py
2 # Created: 08-31-2020
3 # Author: Andrew Davis
4 # Purpose: A simple script for lecture
5
6 # Part 1: Gather input from the user interactively and store it for later reuse.
7 name = input("Enter your name: ")
8 age = int(input("Enter your age: "))
9
10 # Part 2: Output results by concatenating strings with variable values
11 # that store user data.
12 print("Hello, " + name + ".")
13 print("In 4 years, you will be " + str(age + 4) + " years old.")
```

1. Give meaningful names to variables.
2. Use space around operators (e.g, =, + )
3. Use comments at the top of file
4. Organize code in “blocks” of related statements preceded by comments for block.
5. Use space between blocks to improve readability.

# Error messages in Python

## Type Errors

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

`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



# Test your knowledge

1. Create simple **expressions** that combine **values** of different **types** and math **operators**.
2. Which operators can be used with **string values**? Give examples of expressions involving them. What happens when you use other operators?
3. Write a few **assignment statements**, using as assigned values either **literals** or expressions. Experiment with different **variable names** that start with different characters to learn what is allowed and what not.
4. Perform different **function calls** of the **built-in functions**: max, min, len, type, int, str, float, round, print.
5. Create **complex expressions** that combine variables, function calls, operators, and literal values.
6. Use the function **print** to display the result of expressions involving string and numerical values.
7. Write simple examples that use **input** to collect values from a user and use them in simple expressions. Remember to **convert** numerical values.
8. Create situations that raise different kinds of **errors**: Type, Value, Name, or Syntax errors.