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)
  `'abc' + 'def' -> 'abcdef'`

Simple Expressions: Python as calculator

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

```
Input          Output
Expressions    Values
In [...]       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  ┘
```

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'
'rocks!'                'rocks!'

'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

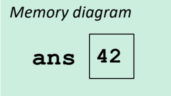
- A variable is a way to remember a value for later in the computer's memory.
- A variable is created by an **assignment statement**, whose form is

  `varName = expression`

  `Example:` `ans = 42` *# ans is the varName, 42 is the expression saved in ans*

  This line of code is executed in two steps:

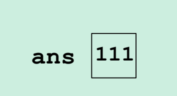
  1. Evaluate ***`expression`*** to its value ***`val`***
  2. If there is no variable box already labeled with ***`varName,`*** create a new box labeled with ***`varName`*** and store ***`val`*** in it; otherwise, change the contents of the existing box labeled ***`varName`*** to ***`val`*** .



Memory Diagram Model: Variable as a Box

- How does the memory diagram change if we evaluate the following expression?

  `ans = 2*ans+27`



- The expression checks the most recent ***`val`*** of **`ans`** (42), re-evaluates the new expression based on that value, and reassigns the value of **`ans`** accordingly.
- **`ans = 2*42+27`**
- **`ans = 111`**

Variable summary

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"!

Variable Examples

Concepts in this slide: variables, assignment statement, memory

In […]	Memory Diagram	Out […]	Notes
`fav = 17`	fav 17		Assignment statements makes box, no output
`fav`		17	Returns current contents of fav
`fav + fav`		34	The contents of fav are unchanged
`lucky = 8`	lucky 8		Makes new box, has no output
`fav + lucky`		25	Variable contents unchanged
`aSum = fav + lucky`	aSum 25		Makes new box, has no output
`aSum * aSum`		625	Variable contents unchanged

Variable Examples

Concepts in this slide: variables, assignment statement, memory

How does the memory diagram change when we change the values of our existing variables? How are strings stored in memory?

In [...]	Memory Diagram	Out [...]	Notes
`fav = 11`	fav 11		Change contents of fav box to 11
`fav = fav - lucky`	fav 3		Change contents of fav box to 3
`name = 'CS111'`	name 'CS111'		Makes new box containing string. Strings are drawn *outside* box with arrow pointing to them (b/c they're often "too big" to fit inside box
`name*fav`		`'CS111CS111CS111'`	string*int will repeat the string int # of times

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

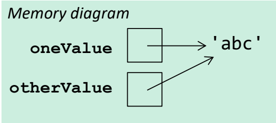
Concepts in this slide: Values can have multiple names. Functions are also values.

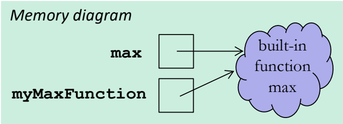
One value can have multiple names. These names refer to the same value in the computer memory. See the examples below for variables and functions.

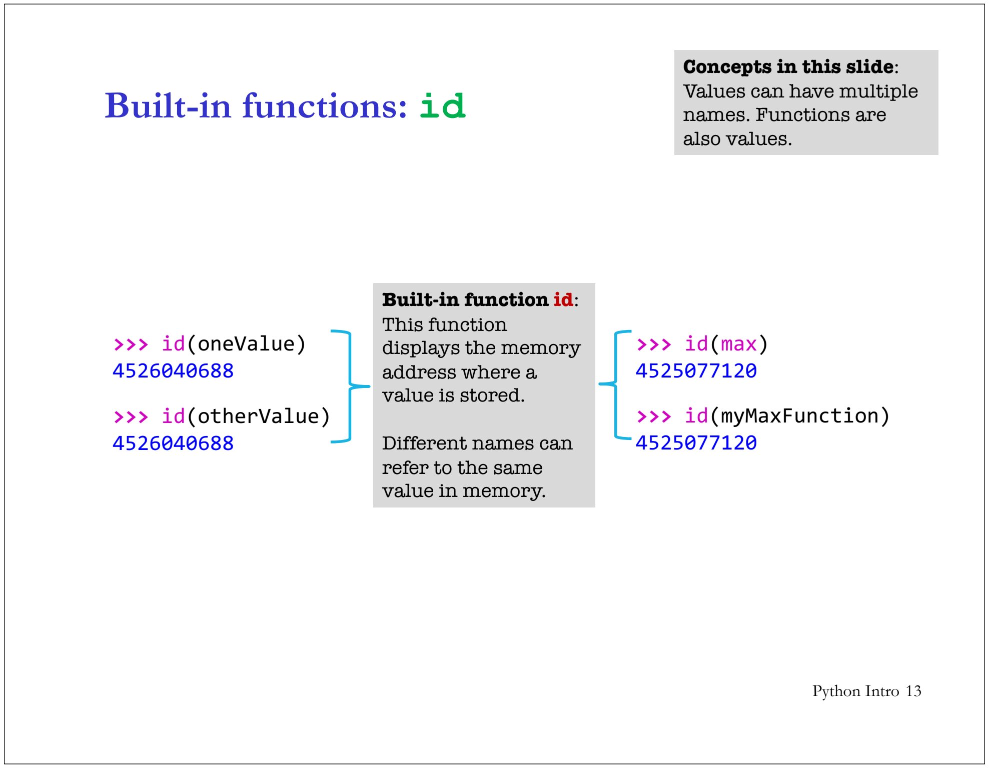
```
>>> oneValue = 'abc'

>>> 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

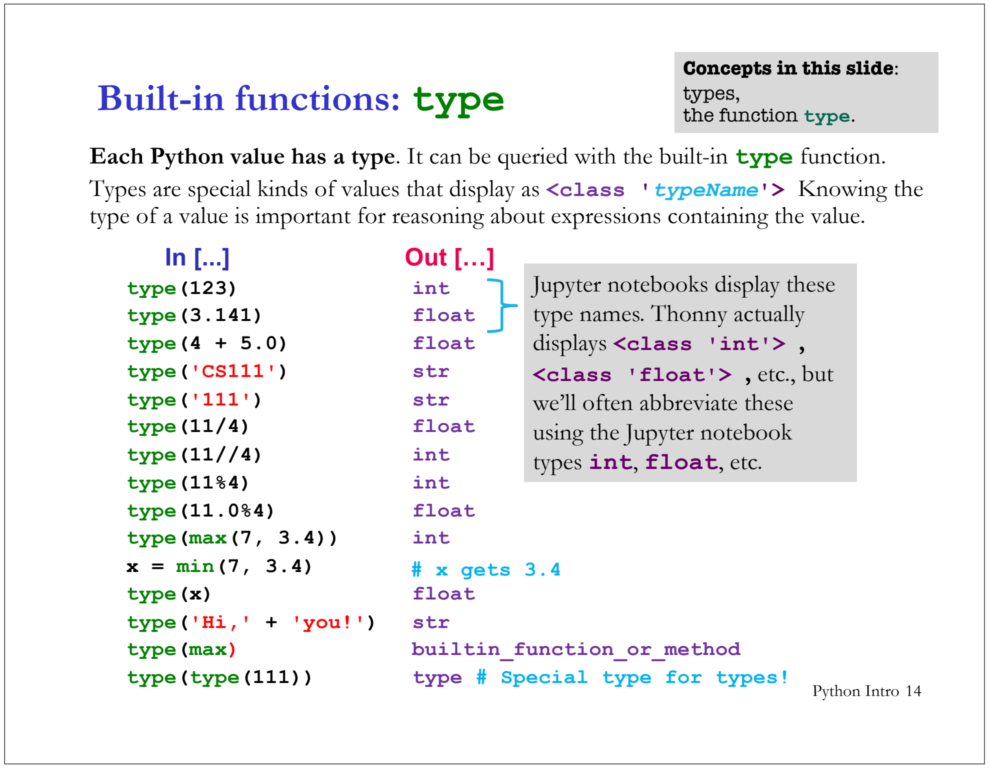
Concepts in this slide: Values can have multiple names. Functions are also values.

```
>>> 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

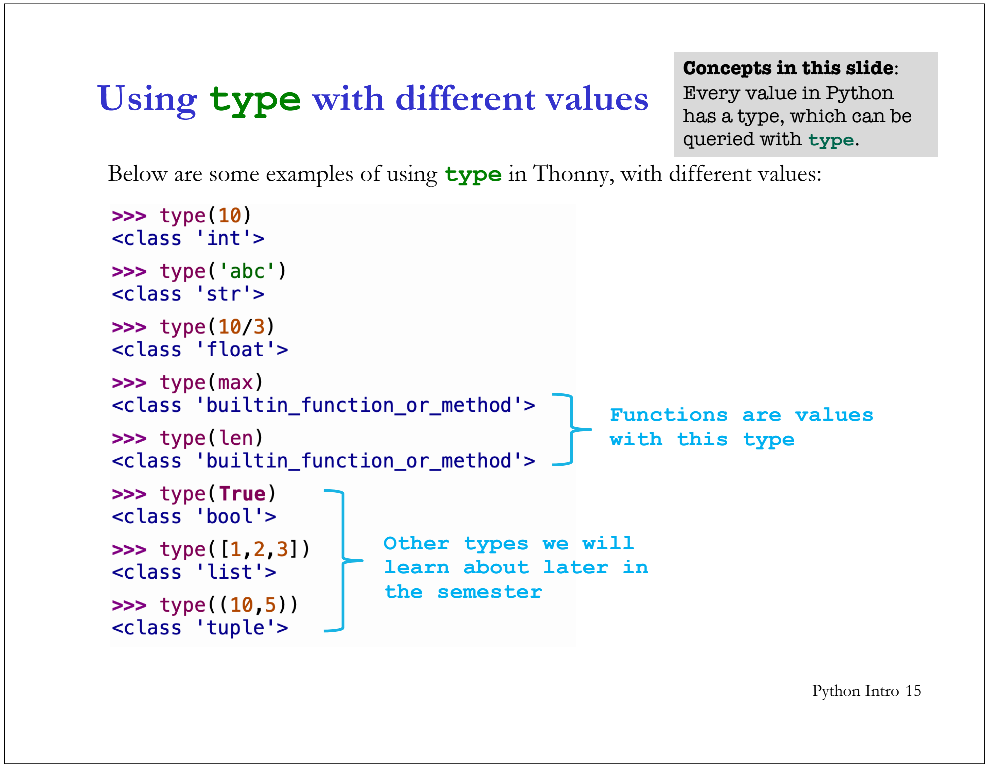
Concepts in this slide: types, the function type.

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(max)	builtin_function_or_method
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

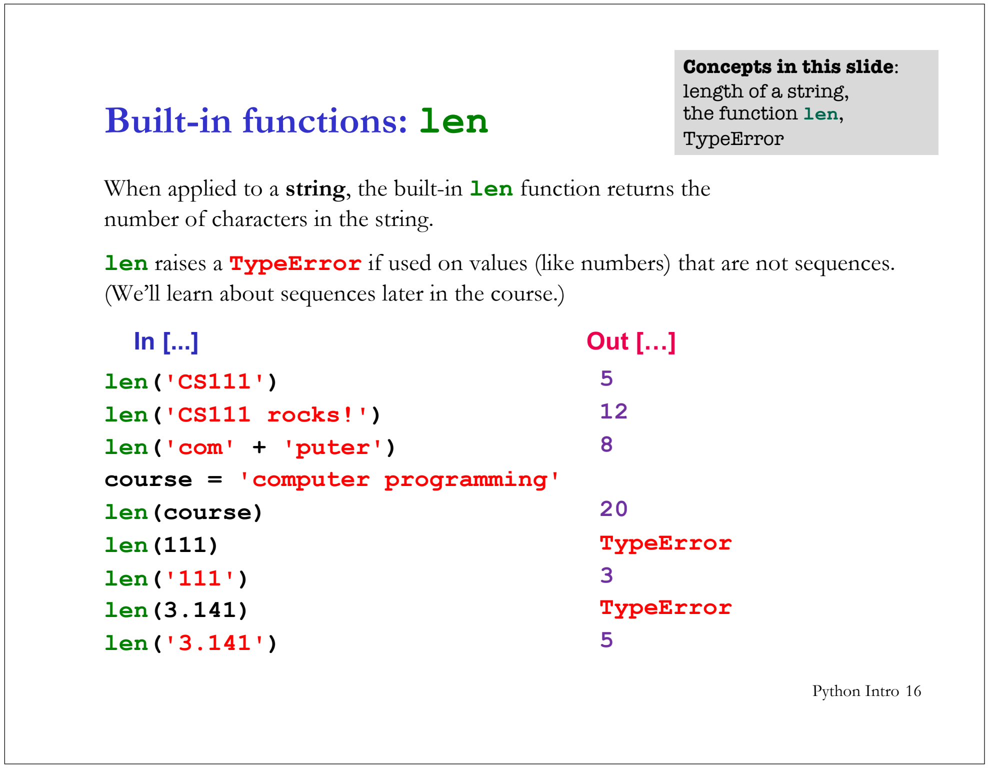
Concepts in this slide: Every value in Python has a type, which can be queried with type.

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

Concepts in this slide: the `str` function

Built-in functions: `str`

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
```

Concepts in this slide: `float` function, ValueError

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 […]
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.(You can learn about such representations in *CS240 Fundamentals of Computer Systems*.)

```
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 ┐
round(4.5)             5 ┘ # always rounds up for 0.5
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 ┐
round(2.3 - 2.0, 1)    0.3 ┘ # Compare to previous slide
```

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, **which is not displayed in Thonny or Jupyter.** Note that **print** also does **not** display any quotation marks for strings.

```
Input statements                     Characters displayed in
In [...]                             console (*not* the output
                                     value of the expression!)

print(7)                             7

print('CS111')                       CS111

print(len(str('CS111')) *            15
min(17,3))

college = 'Wellesley'                I go to Wellesley
print('I go to ' + college)

dollars = 10                         The movie costs $10.
print('The movie costs $'
      + str(dollars) + '.')
```

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 [...]                              Console

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

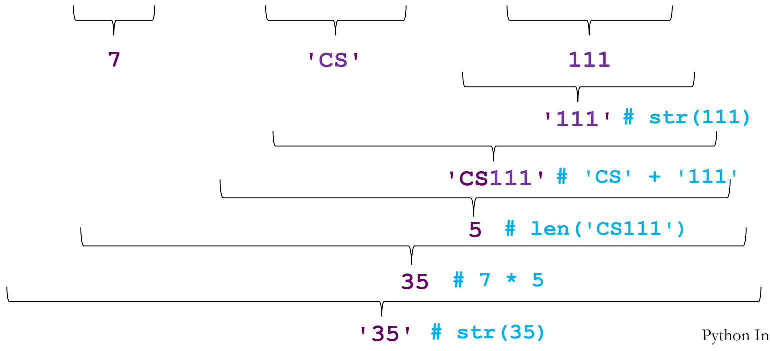
Complex Expression Evaluation

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)
```

More print examples

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

```
In [4]: print('one\ntwo\three') # '\n' is a single special
one                              # newline character.
two                              # Printing it causes the
three                            # display to shift to the
                                 # next line.

In [5]: print('one', 'two', 'three', sep='\n')
one                              # Like previous example,
two                              # but use sep keyword arg
three                            # for newlines

In [6]: str(print(print('CS'), print('CS')))
CS # printed by 2nd print
111 # printed by 3rd print.
None None # printed by 1st print; shows that print returns None

Out [6]: 'None' # result 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 [7]: input('Enter your name: ')
Enter your name: Olivia Rodrigo
```

Magenta text is entered by user.

Brown text is prompt.

```
Out [7]: 'Olivia Rodrigo'
```

Built-in functions: `input`

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

```
In [8]: age = input('Enter your age: ')
Enter your age:20
```

No output from assignment.

```
In [9]: age
Out [9]: '20'
```

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

```
In [10]: 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 [11]: age = float(input('Enter your age: '))
Enter your age: 18
```

Example of nested function calls.

```
In [12]: age + 4
Out [12]: 22.0
```

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

Concepts in this slide:
Expressions, statements

Expressions vs. Statements

Phrases that **produce a value**. E.g. :

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

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

Phrases that **perform an action / change the state of the program** (can be visible, invisible, or both):

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

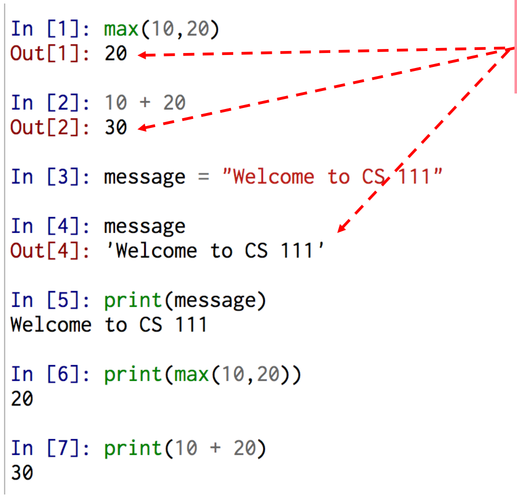
Statements may contain expressions, which are evaluated before the action is performed.

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

We'll consider expressions that return a `None` value to be kinds of statements. Recall that `None` is not normally displayed in Thonny or Jupyter.

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`

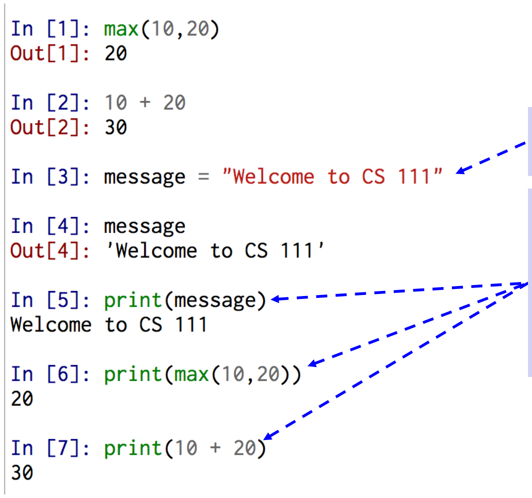


```
In [1]: max(10,20)
Out[1]: 20

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
```

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`



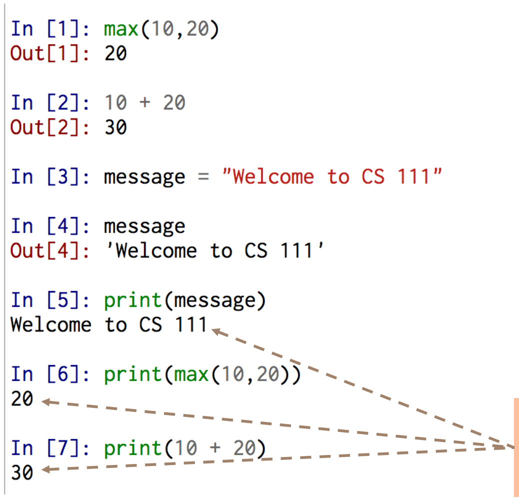
```
In [1]: max(10,20)
Out[1]: 20

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`

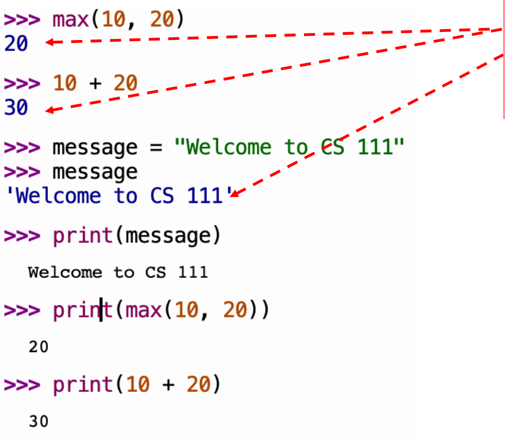


```
In [1]: max(10,20)
Out[1]: 20

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 Thonny

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 Thonny

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 Thonny

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

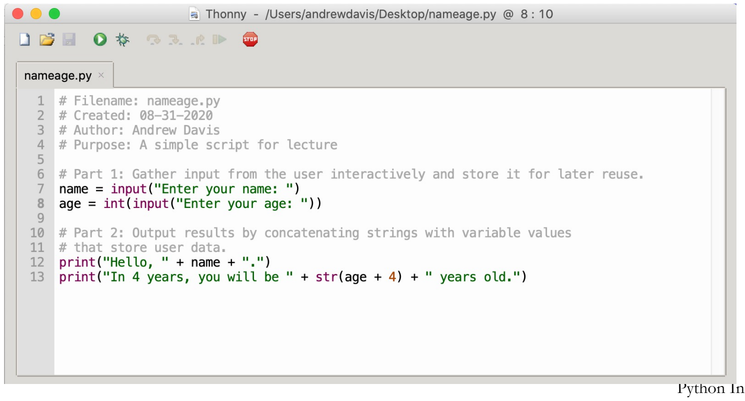
```
>>> max(10, 20)
20
>>> 10 + 20
30
>>> message = "Welcome to CS 111"
>>> message
'Welcome to CS 111'
>>> print(message)
  Welcome to CS 111
>>> print(max(10, 20))
  20
>>> print(10 + 20)
  30
```

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.

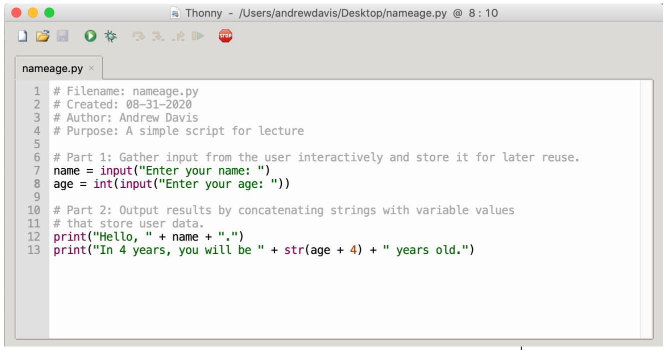
Rather than interactively entering code into the **Python Shell**, we can enter it in the **Editor Pane**, where we can edit it and save it away as a file with the **.py** extension (a Python program). Here is a **nameage.py** program. Lines beginning with **#** are comments We run the program by pressing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



```
# Filename: nameage.py
# Created: 08-31-2020
# Author: Andrew Davis
# Purpose: A simple script for lecture

# Part 1: Gather input from the user interactively and store it for later reuse.
name = input("Enter your name: ")
age = int(input("Enter your age: "))

# Part 2: Output results by concatenating strings with variable values
# that store user data.
print("Hello, " + name + ".")
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.



```
# Filename: nameage.py
# Created: 08-31-2020
# Author: Andrew Davis
# Purpose: A simple script for lecture

# Part 1: Gather input from the user interactively and store it for later reuse.
name = input("Enter your name: ")
age = int(input("Enter your age: "))

# Part 2: Output results by concatenating strings with variable values
# that store user data.
print("Hello, " + name + ".")
print("In 4 years, you will be " + str(age + 4) + " years old.")
```

1. Lines should not be longer than 80 characters
2. Give meaningful names to variables.
3. Use space around operators (e.g, **=**, **+**)
4. Use comments at the top of file
5. Organize code in “blocks” of related statements preceded by comments for block.
6. Use space between blocks to improve readability.
7. For CS111 coding style guidelines, see **http://cs111.wellesley.edu/reference/styleguide**

Error messages in Python

Concepts in this slide:
Error types,
Error messages.

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
```

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`**, **`id`**, **`type`**, **`len`**, **`str`**, **`int`**, **`float`**, **`round`**.
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**: **`TypeError`**, **`ValueError`**, **`NameError`**, and **`SyntaxError`**.