[[](https://www.coursera.org/learn/learn-to-program/resources/oTgXi)](https://www.coursera.org/learn/learn-to-program/resources/oTgXi)

* [Chapter 2.2. Expressions and Values: Arithmetic in Python](https://www.coursera.org/learn/learn-to-program/resources/oTgXi)
* [Chapter 2.3. What Is a Type?](https://www.coursera.org/learn/learn-to-program/resources/oTgXi)
* [Chapter 2.6. How Python Tells You Something Went Wrong](https://www.coursera.org/learn/learn-to-program/resources/oTgXi)
* *[Optional reading](https://www.coursera.org/learn/learn-to-program/resources/oTgXi)*

**Python as a Calculator**

**Arithmetic Operators**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operator | Operation | Expression | English description | Result |
| + | addition | 11 + 56 | 11 plus 56 | 67 |
| - | subtraction | 23 - 52 | 23 minus 52 | -29 |
| \* | multiplication | 4 \* 5 | 4 multiplied by 5 | 20 |
| \*\* | exponentiation | 2 \*\* 5 | 2 to the power of 5 | 32 |
| / | division | 9 / 2 | 9 divided by 2 | 4.5 |
| // | integer division | 9 // 2 | 9 divided by 2 | 4 |
| % | modulo (remainder) | 9 % 2 | 9 mod 2 | 1 |

**Types int and float**

A *type* is a set of values and operations that can be performed on those values.

Two of Python's numeric types:

* int: integer  
  For example: 3, 4, 894, 0, -3, -18
* float: floating point number (an approximation to a real number)  
  For example: 5.6, 7.342, 53452.0, 0.0, -89.34, -9.5

**Arithmetic Operator Precedence**

When multiple operators are combined in a single expression, the operations are evaluated in order of precedence.

|  |  |
| --- | --- |
| Operator | Precedence |
| \*\* | highest |
| - (negation) |  |
| \*, /, //, % |  |
| + (addition), - (subtraction) | lowest |

**Syntax and Semantics**

*Syntax*: the rules that describe valid combinations of Python symbols

*Semantics*: the meaning of a combination of Python symbols is the meaning of an instruction — what a particular combination of symbols does when you execute it.

**Errors**

A syntax error occurs when we an instruction with invalid syntax is executed. For example:

>>> 3) + 2 \* 4

SyntaxError: invalid syntax

A semantic error occurs when an instruction with invalid semantics is executed. For example:

>>> 89.4 / 0

Traceback (most recent call last):

File "", line 1, in

89.4 / 0

ZeroDivisionError: float division by zero

Jennifer Campbell • Paul Gries  
University of Toronto

# Python and Computer Memory

### Computer Memory

For the purpose of this course, you may think of computer memory as a long list of storage locations where each location is identified with a unique number and each location houses a value. This unique number is called a memory address. Typically, we will write memory addresses as a number with an "id" as a prefix to distinguish them from other numbers (for example, id201 is memory address 201).

Variables are a way to keep track of values stored in computer memory. A variable is a named location in computer memory. Python keeps variables in a separate list from values. A variable will contain a memory address, and that memory address contains the value. The variable then refers to the value. Python will pick the memory addresses for you.

### Terminology

A value has a memory address.  
A variable contains a memory address.  
A variable refers to a value.  
A variable points to a value.

Example: Value 8.5 has memory address id34.  
Variable shoe\_size contains memory address id34.  
The value of shoe\_size is 8.5.  
shoe\_size refers to value 8.5.  
shoe\_size points to value 8.5.

**Variables**

**Assignment statements**

The general form of an assignment statement:

*variable = expression*

Example assignment statements:

>>> base = 20

>>> height = 12

>>> area = base \* height / 2

>>> area

120.0

The rules for executing an assignment statement:

1. Evaluate the expression. This produces a memory address.
2. Store the memory address in the variable.

**Variable names**

The rules for legal Python names:

1. Names must start with a letter or \_.
2. Names must contain only letters, digits, and \_.

For Python, in most situations, the convention is to use pothole\_case.

# Visualizing Assignment Statements

On the Resources page is a link to a Python Visualizer that follows the model we use to draw pictures of computer memory.

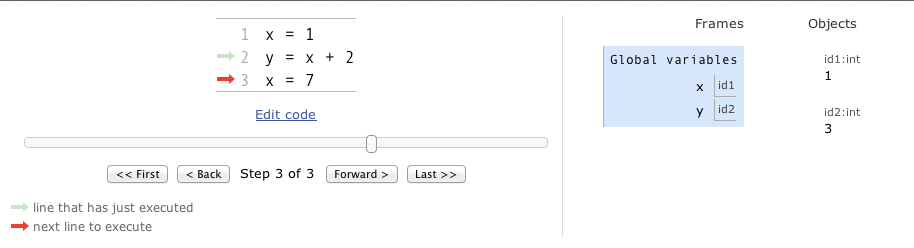
Consider this code:

x = 1

y = x + 2

x = 7

When we trace this in the visualizer and click button Forward twice, this is the result:



Clicking Forward once more results in this:



Notice that y's value did not change during this step.

Here is [a link to the Python Visualizer](http://tinyurl.com/mpy3uxn) containing this code so that you can explore this yourself. **We strongly encourage you to step forward and backward through this program until you understand every step of execution.**

**Built-in Functions**

**Function Call**

The general form of a function call:

*function\_name(arguments)*

The rules for executing a function call:

1. Evaluate the arguments.
2. Call the function, passing in the argument values.

Terminology:

* *Argument*: a value given to a function
* *Pass*: to provide to a function
* *Call*: ask Python to evaluate a function
* *Return*: pass back a value

Example function calls:

>>> abs(-23)

23

>>> abs(56.24)

56.24

**Function dir**

Python has a set of built-in functions. To see the list of built-in functions, run dir(\_\_builtins\_\_):

>>> dir(\_\_builtins\_\_)< br/> ['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '\_', '\_\_build\_class\_\_', '\_\_debug\_\_', '\_\_doc\_\_', '\_\_import\_\_', '\_\_name\_\_', '\_\_package\_\_', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']

**Function help**

To get information about a particular function, call help and pass the function as the argument. For example:

>>> help(abs)

Help on built-in function abs in module builtins:

abs(...)

abs(number) -> number

Return the absolute value of the argument.

**Optional arguments**

In the description of function pow below, the square brackets around [, z] indicate that the third argument is optional:

>>> help(pow)

Help on built-in function pow in module builtins:

pow(...)

pow(x, y[, z]) -> number

With two arguments, equivalent to x\*\*y. With three arguments,

equivalent to (x\*\*y) % z, but may be more efficient (e.g. for longs).

Function pow can be called with either two or three arguments:

>>> pow(2, 5)

32

>>> pow(2, 5, 3)

2

**Defining Functions**

**Function Definitions**

The general form of a function definition:

def function\_name(parameters):

body

* def: a keyword indicating a function definition
* function\_name: the function name
* parameters:
  + the parameter(s) of the function, 0 or more and are separated by a comma
  + a parameter is a variable whose value will be supplied when the function is called
* body:
  + 1 or more statements, often ending with a return statement

Example of a function definition:

def f(x):

return x \*\* 2

**return statement**

The general form of a return statement:

return expression

The rules for executing a return statement:

1. Evaluate the expression. This produces a memory address.
2. Pass back that memory address to the caller. Exit the function.

**Function Calls**

Function calls are expressions and the result can be stored in a variable.

The general form of a function call:

function\_name(arguments)

The rules for executing a function call:

1. Evaluate the arguments to produce memory addresses.
2. Store those memory addresses in the corresponding parameters.
3. Execute the body of the function.

Example of a function definition and function calls:

>>> def area(base, height):

return base \* height / 2

>>> area(3, 4)

6.0

>>> result = area(10, 7.45)

>>> result

37.25

**Saving your programs to ".py" files**

We usually save our Python programs in ".py" files. A file can contain multiple function definitions and other statements. Before calling a function from a ".py" file in the shell in IDLE, you need to first execute Run -> Run Module, or else the shell will not recognize the function call.

# Type str: Strings in Python

### String Literal

A *string literal* is a sequence of characters. In Python, this type is called str. Strings in Python start and end with a single quotes (') or double quotes ("). A string can be made up of letters, numbers, and special characters. For example:

>>> 'hello'

'hello'

>>> 'how are you?'

'how are you?'

>>> 'short- and long-term'

'short- and long-term'

If a string begins with a single quote, it must end with a single quote. The same applies to double-quoted strings. You can not mix the type of quotes.

### Escape Sequences

To include a quote within a string, use an escape character (\) before it. Otherwise Python interprets that quote as the end of a string and an error occurs. For example, the following code results in an error because Python does not expect anything to come after the second quote:

>>> storm\_greeting = 'wow, you're dripping wet.'

SyntaxError: invalid syntax

The escape sequence \' indicates that the second quote is simply a quote, not the end of the string:

>>> storm\_greeting = 'Wow, you\'re dripping wet.'

"Wow, you're dripping wet."

An alternative approach is to use a double-quoted string when including a single-quote within it, or vice-versa. Single- and double-quoted strings are equivalent. For example, when we used double-quotes to indicate the beginning and end of the string, the single-quote in you're no longer causes an error:

>>> storm\_greeting = "Wow, you're dripping wet."

"Wow, you're dripping wet."

### String Operators

|  |  |  |  |
| --- | --- | --- | --- |
| Expression | Description | Example | Output |
| str1 + str2 | concatenate str1 and str1 | print('ab' + 'c') | abc |
| str1 \* int1 | concatenate int1 copies of str1 | print('a' \* 5) | aaaaa |
| int1 \* str1 | concatenate int1 copies of str1 | print(4 \* 'bc') | bcbcbcbc |

Note: *concatenate* means to join together

The \* and + operands obey by the standard precedence rules when used with strings.

All other mathematical operators and operands result in a TypeError.

**Input/Output and str Formatting**

**Function print**

Python has a built-in function named print that displays messages to the user. For example, the following function call displays the string "hello":

>>> print("hello")

hello

In the output above, notice that hello is displayed without the quotation marks. The quotes are only for Python's internal string formatting and are not seen by the user.

The print function may also be called with a mathematical expression for an argument. Python evaluates the mathematical expression first and then displays the resulting value to the user. For example:

>>> print(3 + 7 - 3)

7

Finally, print can take in more than one argument. Each pair of arguments is separated by a comma and a space is inserted between them when they are displayed. For example:

>>> print("hello", "there")

hello there

**return vs. print**

**Recall:** The general form of a return statement:

return *expression*

When a return statement executes, the expression is evaluated to produce a memory address.

* *What is passed back to the caller?*  
  That memory address is passed back to the caller.
* *What is displayed?*  
  Nothing!

An example of return:

>>> def square\_return(num):

return num \*\* 2

>>> answer\_return = square\_return(4)

>>> answer\_return

16

The general form of a print function call:

print(*arguments*)

When a print function call is executed, the argument(s) are evaluated to produce memory address(es).

* *What is passed back to the caller?*  
  Nothing!
* *What is displayed?*  
  The values at those memory address(es) are displayed on the screen.

An example of print:

>>> def square\_print(num):

print("The square of num is", num \*\* 2)

>>> answer\_print = square\_print(4)

The square num is 16

>>> answer\_print

>>>

**Function input**

The function input is a built-in function that prompts the user to enter some input. The program waits for the user to enter the input, before executing the subsequent instructions. The value returned from this function is *always* a string. For example:

>>> input("What is your name? ")

What is your name? Jen

'Jen'

>>> name = input("What is your name? ")

What is your name? Jen

>>> name

'Jen'

>>> location = input("What is your location? ")

What is your location? Toronto

>>> location

'Toronto'

>>> print(name, "lives in", location)

Jen lives in Toronto

>>> num\_coffee = input("How many cups of coffee? ")

How many cups of coffee? 2

>>> num\_coffee

'2'

**Operations on strings**

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Description | Example | Output |
| str1 + str2 | concatenate str1 and str1 | print('ab' + 'c') | abc |
| str1 \* int1 | concatenate int1 copies of str1 | print('a' \* 5) | aaaaa |
| int1 \* str1 | concatenate int1 copies of str1 | print(4 \* 'bc') | bcbcbcbc |

**Triple-quoted strings**

We have used single- and double- quotes to represent strings. The third string format uses triple-quotes and a triple-quoted string can span multiple lines. For example:

>>> print(''' How

are

you?''')

How

are

you?

**Escape Sequences**

Python has a special character called an *escape character*: \. When the escape character is used in a string, the character following the escape character is treated differently from normal. The escape character together with the character that follows it is an *escape sequence*. The table below contains some of Python's commonly used escape sequences.

|  |  |  |  |
| --- | --- | --- | --- |
| Escape Sequence | Name | Example | Output |
| \n | newline (ASCII linefeed - LF) | print('''How are you?''') | How are you? |
| \t | tab (ASCII horizontal tab - TAB) | print('3\t4\t5') | 3 4 5 |
| \\ | backslash (\) | print('\\') | \ |
| \' | single quote (') | print('don\'t') | don't |
| \" | double quote (") | print("He says, \"hi\".") | He says, "hi". |

# Docstrings and Function help

Built-in function help displays the docstring from a function definition. For example, consider this function:

def area(base, height):

"""(number, number) -> number

Return the area of a triangle with dimensions base

and height.

"""

return base \* height / 2

Calling help on function area produces this output:

>>> help(area)

Help on function area in module \_\_main\_\_:

area(base, height)

(number, number) -> number

Return the area of a triangle with dimensions base

and height.

# Function Design Recipe

### The Six Steps

1. **Examples**
   * What should your function do?
   * Type a couple of example calls.
   * Pick a name (often a verb or verb phrase): What is a short answer to "What does your function do"?
2. **Type Contract**
   * What are the parameter types?
   * What type of value is returned?
3. **Header**
   * Pick meaningful parameter names.
4. **Description**
   * Mention every parameter in your description.
   * Describe the return value.
5. **Body**
   * Write the body of your function.
6. **Test**
   * Run the examples.

### Applying the Design Recipe

**The problem:**

The United States measures temperature in Fahrenheit and Canada measures it in Celsius. When travelling between the two countries it helps to have a conversion function. Write a function that converts from Fahrenheit to Celsius.

1. **Examples**
2. >>> convert\_to\_celsius(32)
3. 0
4. >>> convert\_to\_celsius(212)
5. 100
7. **Type Contract**
8. (number) -> number
10. **Header**
11. def convert\_to\_celsius(fahrenheit):
13. **Description**
14. Return the number of Celsius degrees equivalent to fahrenheit degrees.
16. **Body**
17. return (fahrenheit - 32) \* 5 / 9
19. **Test**
20. Run the examples.

#### Putting it all together:

def convert\_to\_celsius(fahrenheit):

''' (number) -> number

Return the number of Celsius degrees equivalent to fahrenheit degrees.

>>> convert\_to\_ccelsius(32)

0

>>> convert\_to\_celsius(212)

100

'''

return (fahrenheit - 32) \* 5 / 9

# Function Reuse

### Calling functions within other function definitions

**The problem:** Calculate the semi-perimeter of a triangle.

**The approach:** Function semiperimeter calls function perimeter.

def perimeter(side1, side2, side3):

'''(number, number, number) -> number

Return the perimeter of a triangle with sides of length

side1, side2 and side3.

>>> perimeter(3, 4, 5)

12

>>> perimeter(10.5, 6, 9.3)

25.8

'''

return side1 + side2 + side3

def semiperimeter(side1, side2, side3):

'''(number, number, number) -> float

Return the perimeter of a triangle with sides of

length side1, side2 and side3.

>>> semiperimeter(3, 4, 5)

6.0

>>> semiperimeter(10.5, 6, 9.3)

12.9

'''

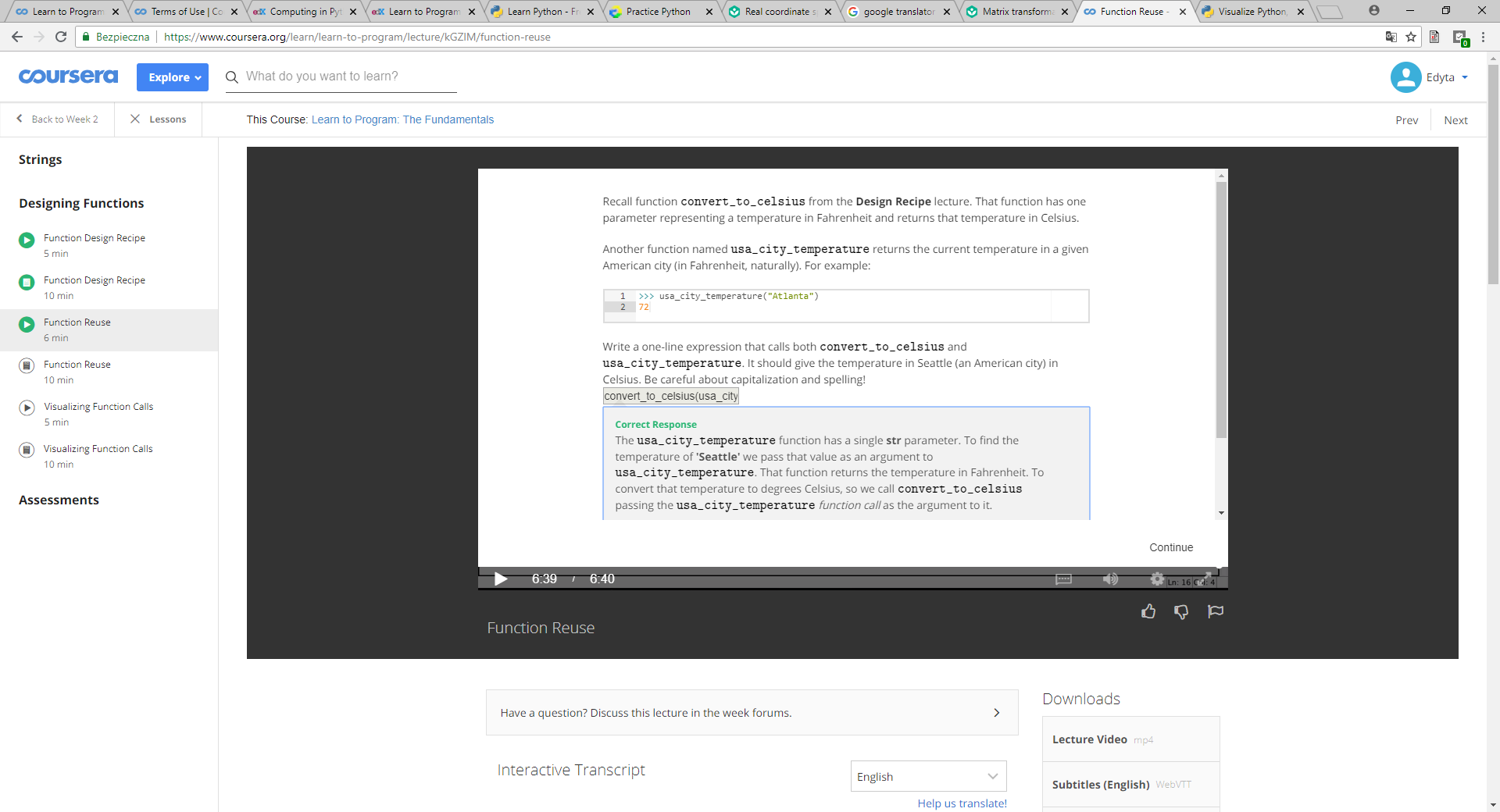
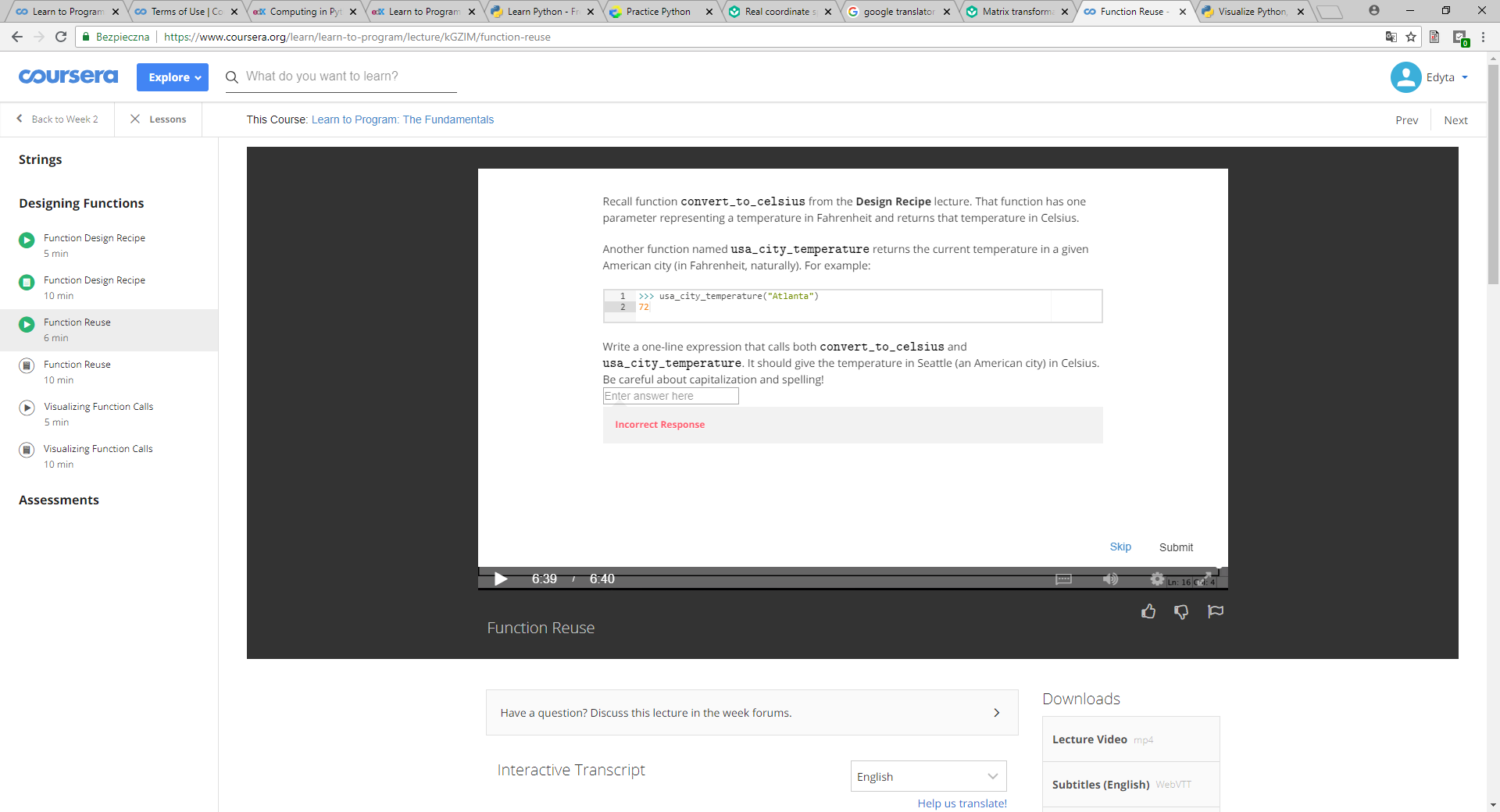
**return perimeter(side1, side2, side3) / 2**

### Calling functions within other function calls

**The problem:** One triangle has a base of length 3.8 and a height of length 7.0. A second triangle has a base of length 3.5 and a height of length 6.8. Calculate which of two triangles' areas is biggest.

**The approach:** Pass calls to function area as arguments to built-in function max.

max(area(3.8, 7.0), area(3.5, 6.8))



# Visualizing Function Calls

We can explore how Python manages function calls using the Python Visualizer. (See the Resources page.)

In the example below, function convert\_to\_seconds contains a call on convert\_to\_minutes.

def convert\_to\_minutes(num\_hours):

"""(int) -> int

Return the number of minutes there are in num\_hours hours.

>>> convert\_to\_minutes(2)

120

"""

result = num\_hours \* 60

return result

def convert\_to\_seconds(num\_hours):

"""(int) -> int

Return the number of seconds there are in num\_hours hours.

>>> convert\_to\_seconds(2)

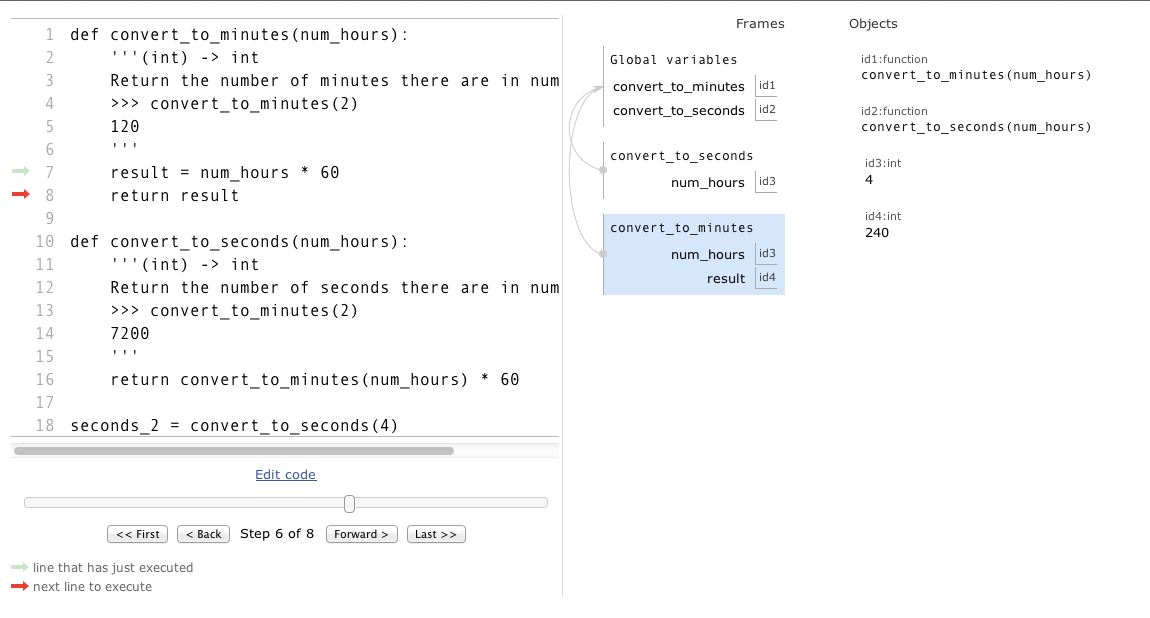
7200

"""

return convert\_to\_minutes(num\_hours) \* 60

seconds\_2 = convert\_to\_seconds(4)

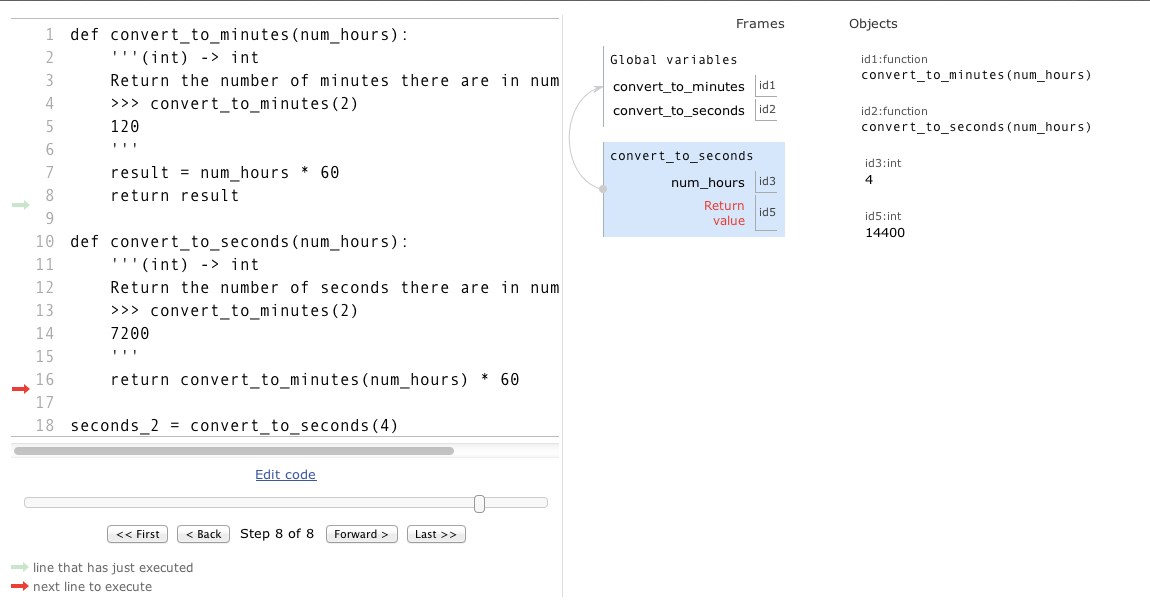
Here is what the memory model looks like just before the return statement inside function convert\_to\_minutes looks like:



Note that there are three stack frames on the call stack: the main one, then underneath that a frame for the call on function convert\_to\_seconds, and underneath that the frame for the call on function convert\_to\_minutes.

Here is [a link to the Python Visualizer](http://tinyurl.com/l7hqbwm) at this stage of the execution so that you can explore this yourself. **We strongly encourage you to step backward and forward through this program until you understand every step of execution.**

When the return statement is executed, the call on convert\_to\_minutes exits. The bottom stack frame is removed, and execution resumes using the stack frame for convert\_to\_seconds:



# Functions, Variables, and the Call Stack

### Understanding Scope

Below is an explanation and review of the example used in the video.

def convert\_to\_minutes(num\_hours):

""" (int) -> int

Return the number of minutes there are in num\_hours hours.

"""

minutes = num\_hours \* 60

return minutes

def convert\_to\_seconds(num\_hours):

""" (int) -> int

Return the number of seconds there are in num\_hours hours.

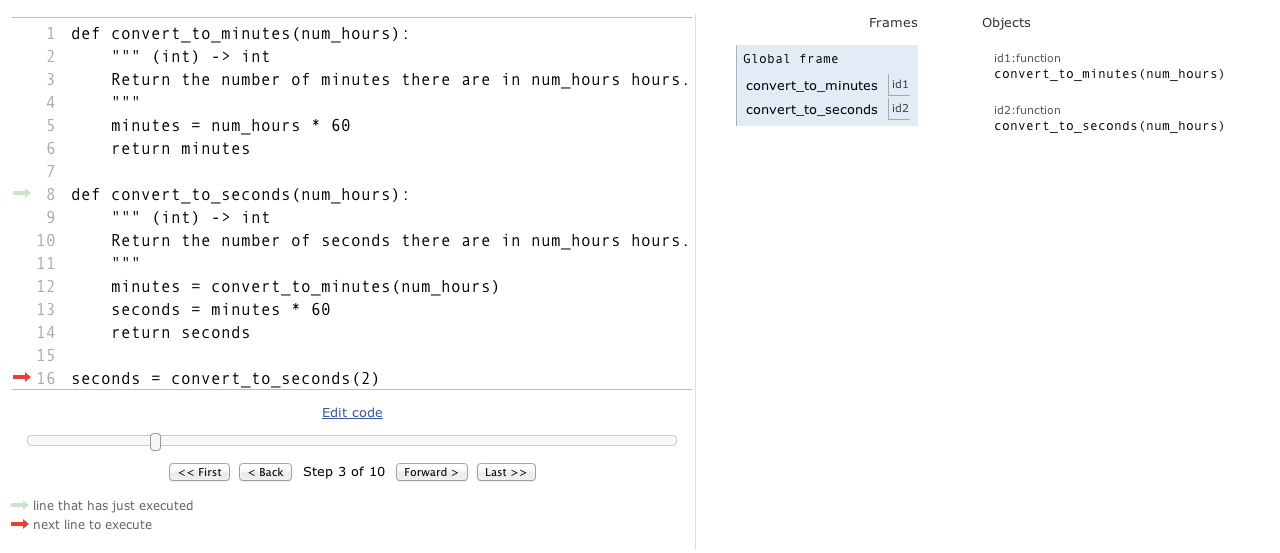
"""

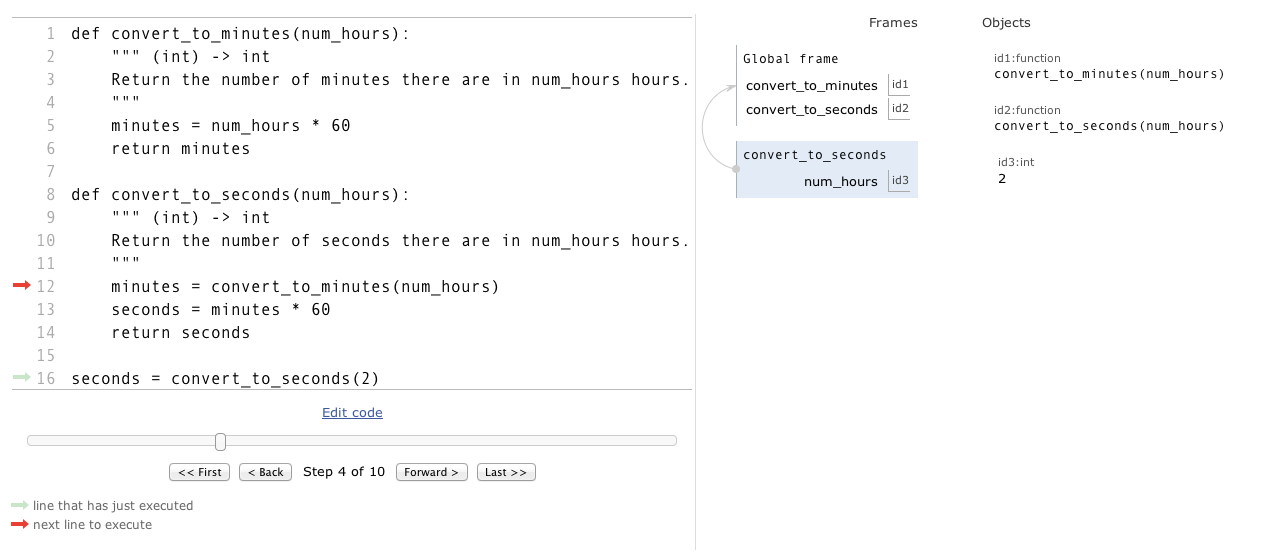
minutes = convert\_to\_minutes(num\_hours)

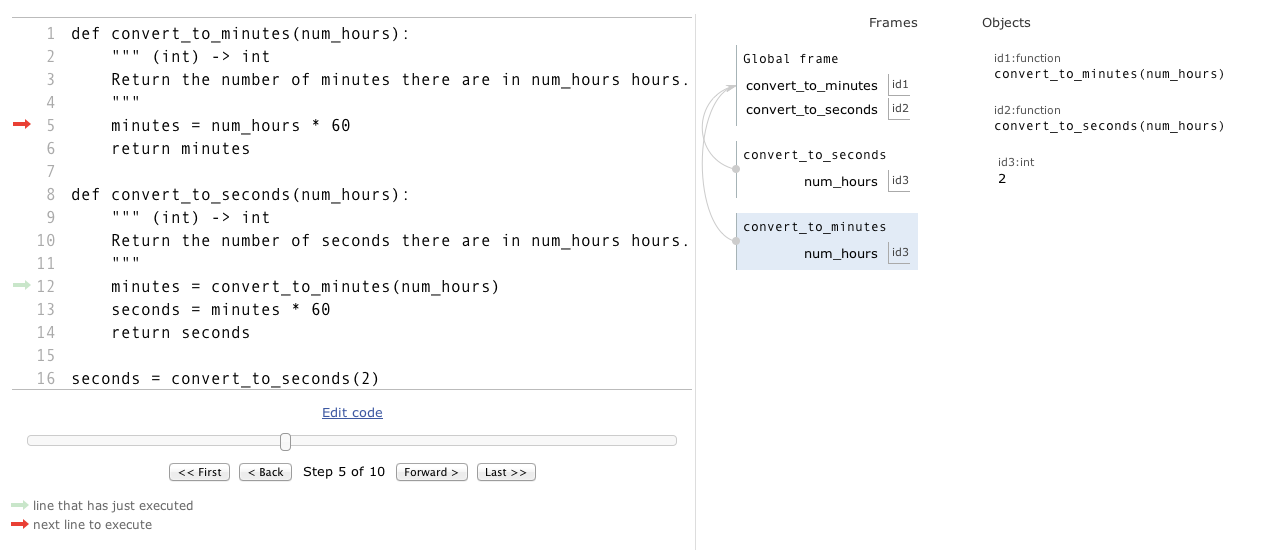
seconds = minutes \* 60

return seconds

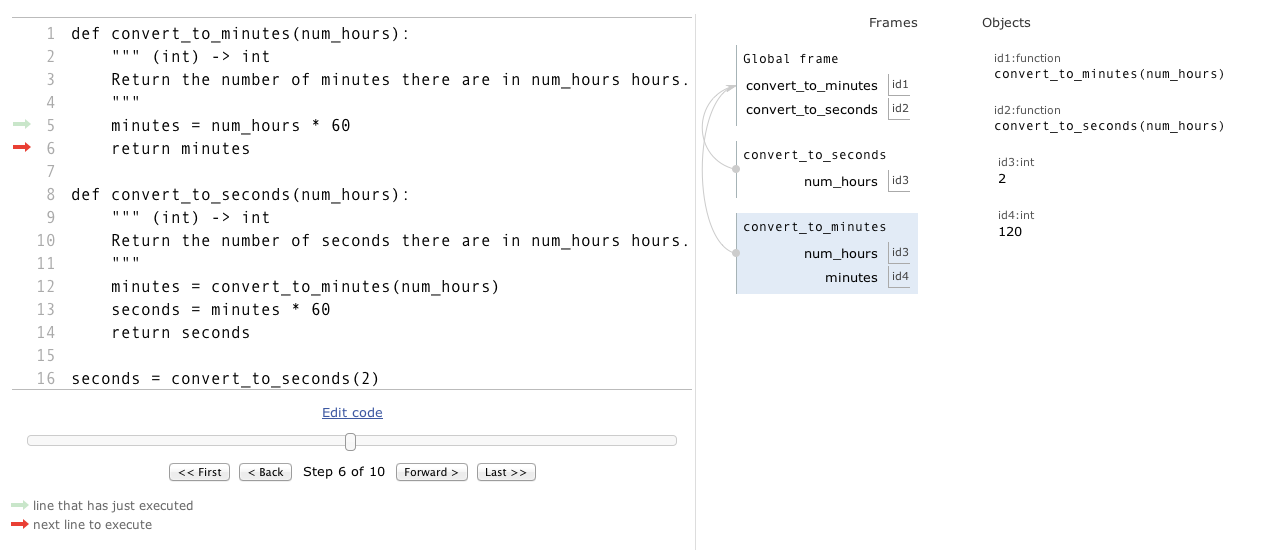
seconds = convert\_to\_seconds(2)

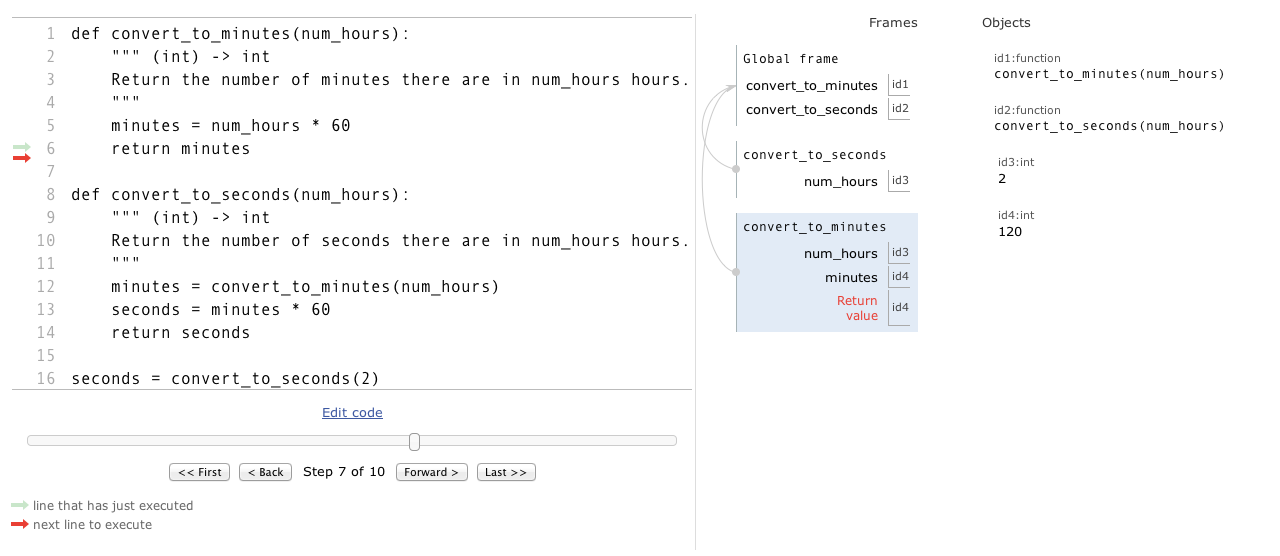
Python defines the first two functions, creates objects for them in the heap, and, in the stack frame for the main program, creates variables that refer to those function objects.  


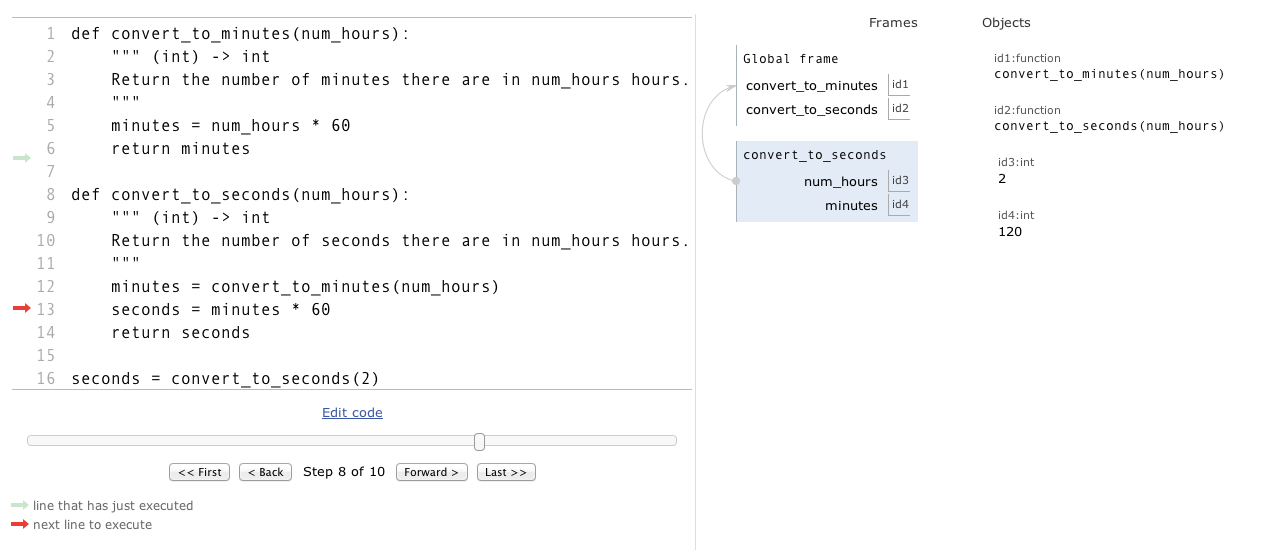
After that, it executes the assignment statement on line 16. The right-hand side of the assignment statement is a function call so we evaluate the argument, 2, first. The frame for convert\_to\_seconds will appear on the call stack. The parameter, num\_hours, will refer to the value 2.  


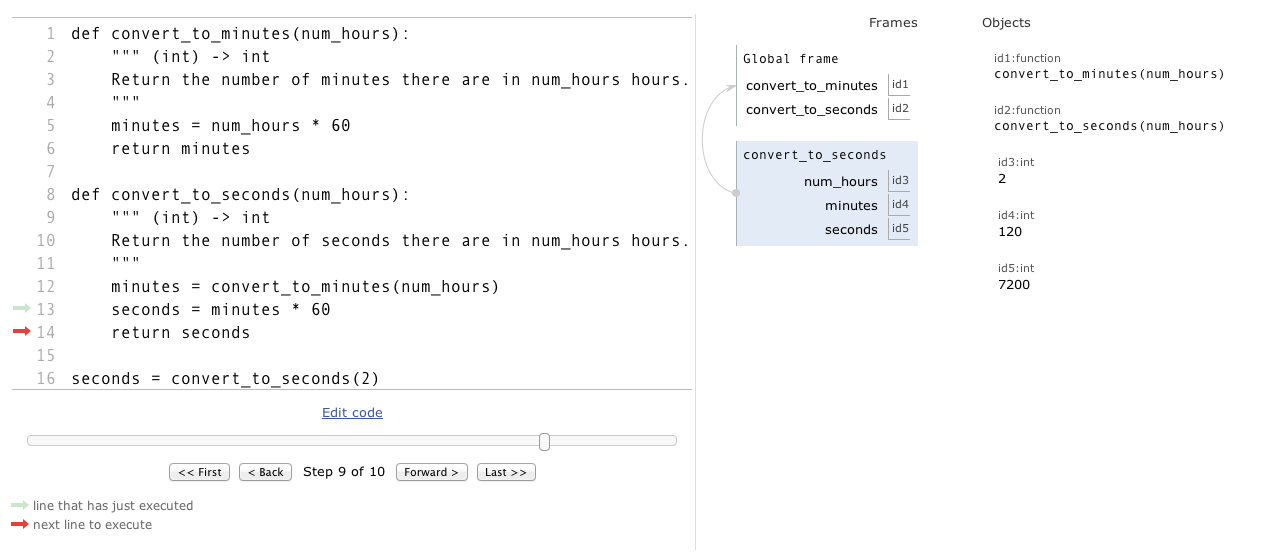
The first statement in function convert\_to\_seconds is an assignment statement. Again, we evaluate the expression on the right-hand side. This is a function call so we evaluate the argument, num\_hours. This produces the value 2. A stack frame for function convert\_to\_minutes is created on the call stack. Python stores the memory address of 2 in the parameter for convert\_to\_minutes, which also happens to be called num\_hours.  


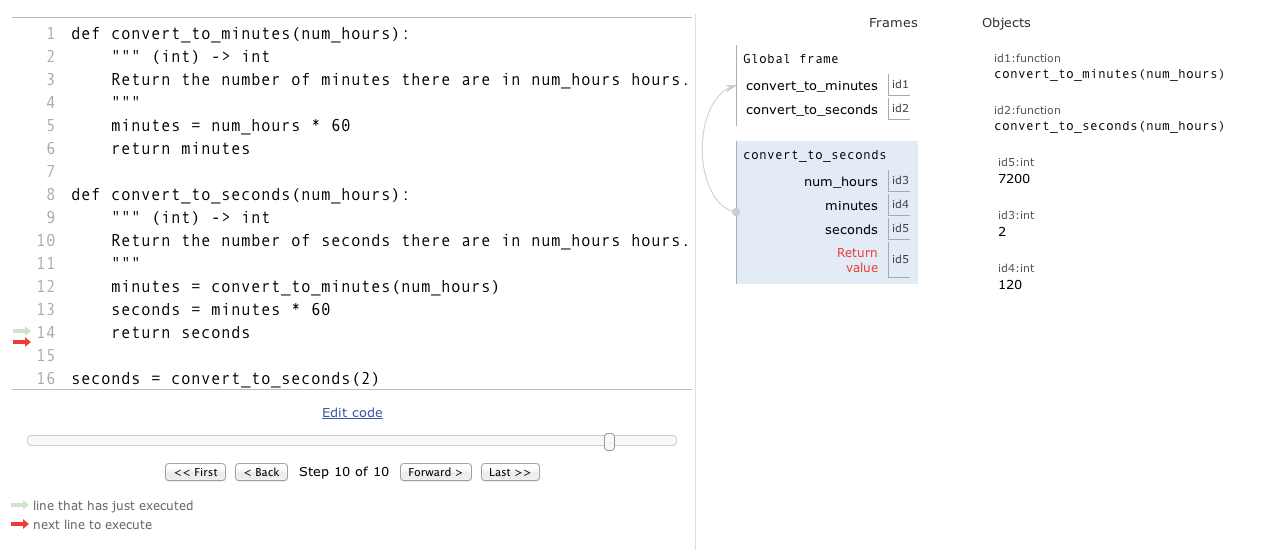
We now see that there are two variables called num\_hours in the call stack; one is in convert\_to\_minutes and the other is in convert\_to\_seconds.

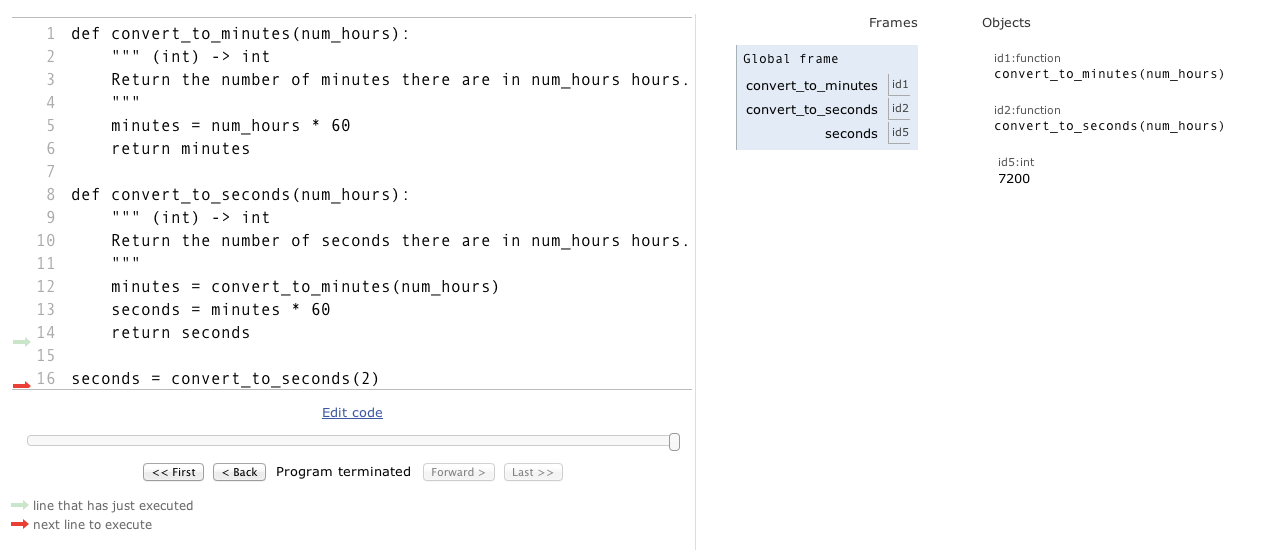
The next line of code Python executes is minutes = num\_hours \* 60. However, which instance of num\_hours will be used? Python always uses the variable in the current stack frame. With an assignment statement, if the variable does not exist in the current stack frame, Python creates it. So, once num\_hours \* 60 is evaluated, variable minutes is created in the current stack frame.  


The last line of the function is return minutes. Once this statement is complete, Python will return to the frame just underneath the top of the call stack.  


So, Python is going to produce the value 120, remove the current stack frame, create a new variable called minutes in the stack frame for convert\_to\_seconds, and store the memory adress of 120 in that variable.  


Python then executes seconds = minutes \* 60. Python evaluates the right-hand side, which produces 7200, and stores the memory address of that value in variable seconds. Since this variable does not exist yet, Python creates it in the current stack frame. 

Next is a return statement. Like we saw above, that is going to return control back to the the main module.  


Once the frame for convert\_to\_seconds is removed, the assignment statement on line 16 (which has been paused a long time!) is completed, and a new variable seconds is created in the stack frame for the main program.  


### Notes and assignment and return statements

**Assignment statement and computer memory**

variable = expression

If a variable does not exist in the current stack frame, Python creates it.

**Return statement and computer memory**

return expression

In addition to evaluating the expression and yielding its value, return also erases the stack frame on top of the call stack.

# Type bool: Booleans in Python

### Boolean values

The Python type bool has two values: True and False.

### Comparison operators

The comparison operators take two values and produce a Boolean value.

|  |  |  |  |
| --- | --- | --- | --- |
| Description | Operator | Example | Result of example |
| less than | < | 3 < 4 | True |
| greater than | > | 3 > 4 | False |
| equal to | == | 3 == 4 | False |
| greater than or equal to | >= | 3 >= 4 | False |
| less than or equal to | <= | 3 <= 4 | True |
| not equal to | != | 3 != 4 | True |

### Logical operators

There are also three logical operators that produce Boolean values: and, or, and not.

|  |  |  |  |
| --- | --- | --- | --- |
| Description | Operator | Example | Result of example |
| not | not | not (80 >= 50) | False |
| and | and | (80 >= 50) and (70 <= 50) | False |
| or | or | (80 >= 50) or (70 <= 50) | True |

### The and Logic Table

The and operator produces True if and only if both expressions are True.

As such, if the first operand is False, the second condition will not even be checked, because it is already known that the expression will produce False.

|  |  |  |
| --- | --- | --- |
| expr1 | expr2 | expr1 and expr2 |
| True | True | True |
| True | False | False |
| False | True | False |
| False | False | False |

### The or Logic Table

The or operator evaluates to True if and only if at least one operand is True.

As such, if the first operand is True, the second condition will not even be checked, because it is already known that the expression will produce True.

|  |  |  |
| --- | --- | --- |
| expr1 | expr2 | expr1 or expr2 |
| True | True | True |
| True | False | True |
| False | True | True |
| False | False | False |

### The not Logic Table

The not operator evaluates to True if and only if the operand is False.

|  |  |
| --- | --- |
| expr1 | not expr1 |
| True | False |
| False | True |

Double-negation can be simplified. For example, the expression not not (4 == 5) can be simplified to 4 == 5.

### Order of Precedence for Logical Operators

The order of precedence for logical operators is: not, and, then or. We can override precedence using parentheses and parentheses can also be added to make things easier to read and understand.

For example, the not operator is applied before the or operator in the following code:

>>> grade = 80

>>> grade2 = 90

>>> not grade >= 50 or grade2 >= 50

True

Parentheses can be added to make this clearer: (not grade >= 50) or (grade2 >= 50)

Alternatively, parentheses can be added to change the order of operations: not ((grade >= 50) or (grade2 >= 50))

# Converting between int, str, and float

### str

Builtin function str takes any value and returns a string representation of that value.

>>> str(3)

'3'

>>> str(47.6)

'47.6'

### int

Builtin function int takes a string containing only digits (possibly with a leading minus sign -) and returns the int that represents. Function int also converts float values to integers by throwing away the fractional part.

>>> int('12345')

12345

>>> int('-998')

-998

>>> int(-99.9)

-99

If function int is called with a string that contains anything other than digits, a ValueError happens.

>>> int('-99.9')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: invalid literal for int() with base 10: '-99.9'

### float

Builtin function float takes a string containing only digits and zero or one decimal points (possibly with a leading minus sign -) and returns the float that represents. Function float also converts int values to floats.

>>> float('-43.2')

-43.2

>>> float('432')

432.0

>>> float(4)

4.0

If function float is called with a string that can't be converted, a ValueError happens.

>>> float('-9.9.9')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: could not convert string to float: '-9.9.9'

# Import: Using Non-Builtin Functions

### Modules

Python contains many functions, but not all of them are immediately available as builtin functions. Instead of being available as builtins, some functions are saved in different modules. A module is a file containing function definitions and other statements.

We may also define our own modules with our own functions.

### import

In order to gain access to the functions in a module, we must import that module.

The general form of an import statement is:

import module\_name

To access a function within a module, we use:

module\_name.function\_name

For example, we can import the Python module math and call the function sqrt from it:

import math

def area2(side1, side2, side3):

semi = semiperimeter(side1, side2, side3)

area = math.sqrt(semi \* (semi - side1) \* (semi - side2) \* (semi - side3))

return area

In addition to importing Python's modules, we can also import the modules that we write. For example, to use the functions from triangle.py (from the video) in another module, we would import triangle. A module being imported should be in the same directory as the module importing it.

# The if statement

If statements can be used to control which instructions are executed. Here is the general form:

if expression1:

body1

[elif expression2: 0 or more clauses

body2]

[else: 0 or 1 clause

bodyN]

elif stands for "else if", so this forms a chain of conditions.

To execute an if statement, evaluate each expression in order from top to bottom. If an expression produces True, execute the body of that clause and then skip the rest open the if statement. If there is an else, and none of the expressions produce True, then execute the body of the else.

For example, given this function:

def report\_status(scheduled\_time, estimated\_time):

""" (float, float) -> str """

if scheduled\_time == estimated\_time:

return 'on time'

elif scheduled\_time > estimated\_time:

return 'early'

else:

return 'delayed'

In the shell:

>>> report\_status(14.3, 14.3)

'on time'

>>> report\_status(12.5, 11.5)

'early'

>>> report\_status(9.0, 9.5)

'delayed'

### A note on None

When execution of a function body ends without having executed a return statement, the function returns value None. The type of None is NoneType.

For example, consider this function:

def report\_status(scheduled\_time, estimated\_time):

""" (float, float) -> str

Return the flight status (on time, early, delayed) for a flight that was

scheduled to arrive at scheduled\_time, but is now estimated to arrive

at estimated\_time.

Pre-condition: 0.0 <= scheduled\_time < 24.0 and 0.0 <= estimated\_time < 24.0

>>> report\_status(14.3, 14.3)

'on\_time'

>>> report\_status(12.5, 11.5)

'early'

>>> report\_status(9.0, 9.5)

'delayed'

"""

if scheduled\_time == estimated\_time:

return 'on time'

In the shell:

>>> report\_status(14,3, 14.3)

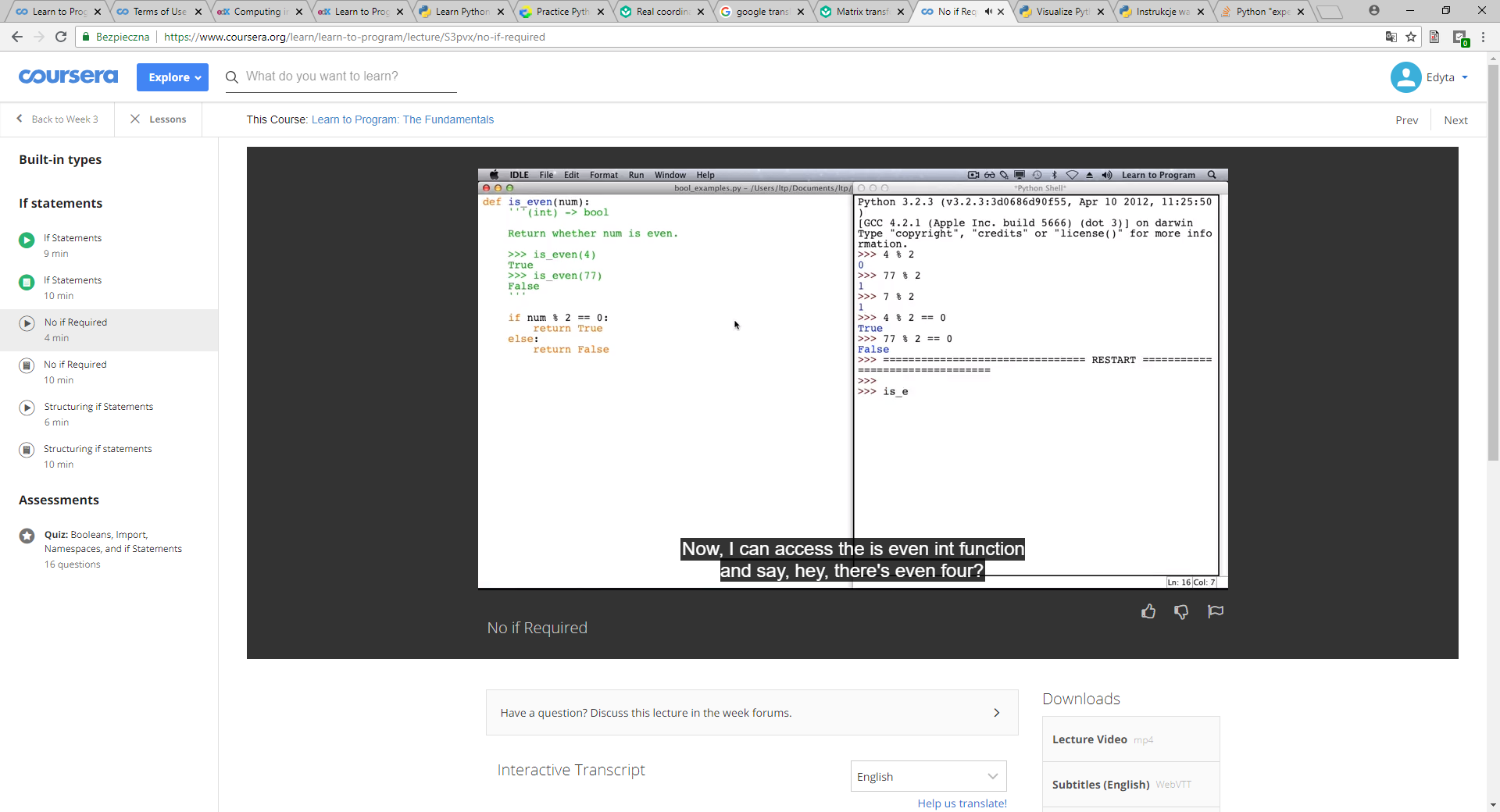
'on time'

>>> report\_status(12.5, 11.5)

>>> print(report\_status(12.5, 11.5))

None

Because the type of None is NoneType, not str, this breaks the Type Contract. To fix this, we would need to complete the rest of the function.



# No if Required

It is common for new programmers to write code like the following:

def is\_even(num):

""" (int) -> bool

Return whether num is even.

"""

if num % 2 == 0:

return True

else:

return False

This works, but is stylistically questionable. It's also more typing and reading than is necessary!

num % 2 == 0 already produces True or False, so that expression can be used with the return statement:

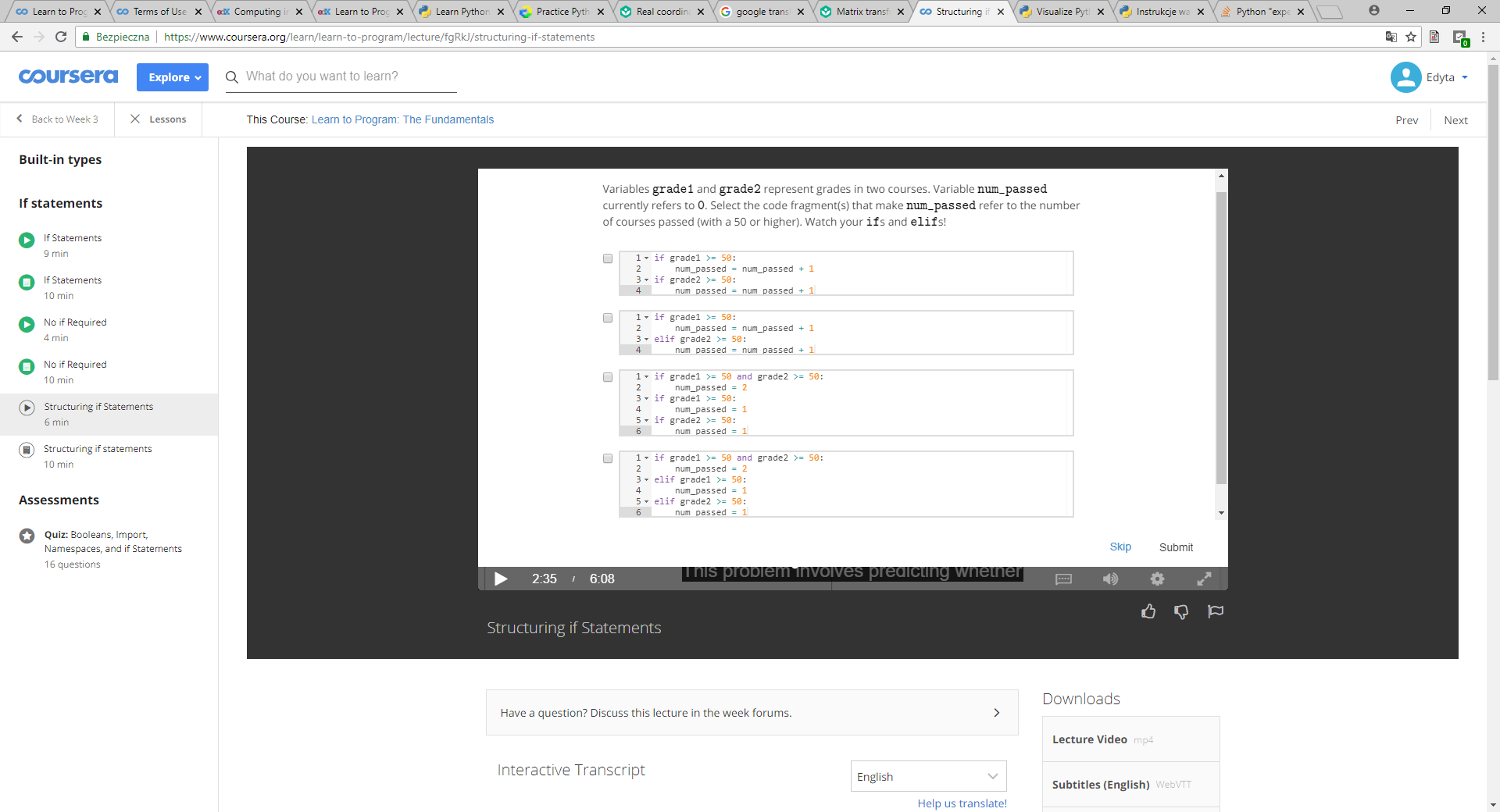
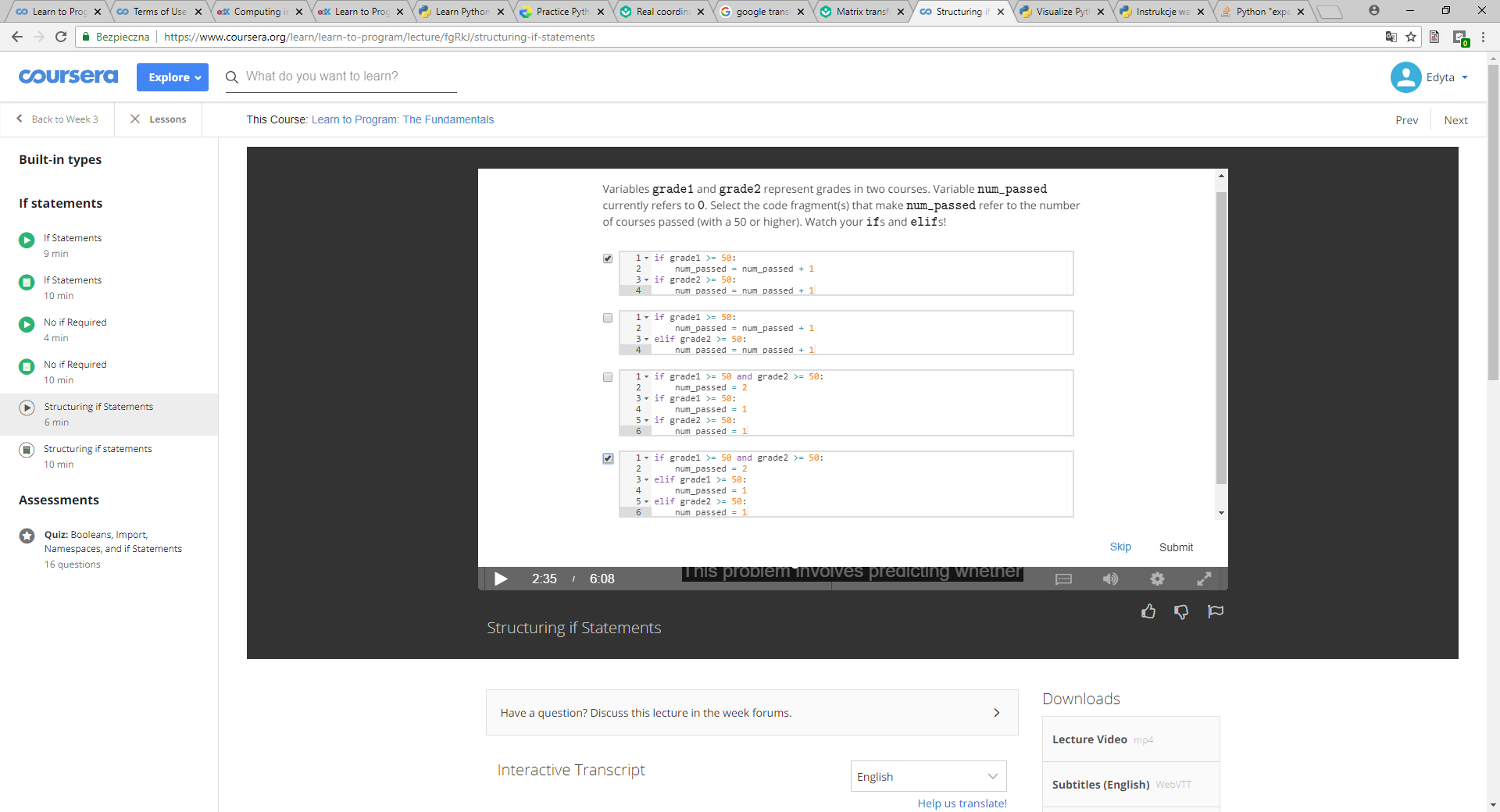
def is\_even(num):

""" (int) -> bool

Return whether num is even.

"""

return num % 2 == 0

  
poprawna odp: 

# Structuring if Statements

### if-elif vs. if-if

An if statement with an elif clause is a single statement. The expressions are evaluated from top to bottom until one produces True or until there are no expressions left to evaluate. When an expression produces True, the body associated with it is executed and then the if statement exits. Any subsequent expressions are ignored. For example:

grade1 = 70

grade2 = 80

if grade1 >= 50:

print('You passed a course with grade: ', grade1)

elif grade2 >= 50:

print('You passed a course with grade: ', grade2)

The if statement condition (grade1 >= 50) evaluates to True, so the body associated with the if is executed and then the if exits. The elif condition is not even evaluated in this case.

It is possible for if statements to appear one after another in a program. Although they are be adjacent to each other, they are completely independent of each other and it is possible for the body of each if to be executed. For example:

grade1 = 70

grade2 = 80

if grade1 >= 50:

print('You passed a course with grade: ', grade1)

if grade2 >= 50:

print('You passed a course with grade: ', grade2)

In the program above, the condition associated with the first if statement (grade1 >= 50) produces True, so the body associated with it is executed. The condition associated with the second if statement (grade2 >= 50) also produces True, so the body associated with it is also executed.

### Nested ifs

It is possible to place an if statement within the body of another if statement. For example:

if precipitation:

if temperature > 0:

print('Bring your umbrella!')

else:

print('Wear your snow boots and winter coat!)

The statement above can be simplified by removing some of the nesting. The message 'Bring your umbrella!' is printed only when both of the if statement conditions are True. The message 'Wear your snow boots and winter coat!' is printed only when the outer if condition is True, but the inner if condition is False. The following is equivalent to the code above:

if precipitation and temperature > 0:

print('Bring your umbrella')

elif precipitation:

print('Wear your snow boots and winter coat!')

# More str Operators

### String Comparisons

The equality and inequlity operators can be applied to strings:

>>> 'a' == 'a'

True

>>> 'ant' == 'ace'

False

>>> 'a' == 'b'

False

>>> 'a' != 'b'

True

We can compare two strings for their dictionary order, comparing them letter by letter:

>>> 'abracadabra' < 'ace'

True

>>> 'abracadabra' > 'ace'

False

>>> 'a' <= 'a'

True

>>> 'A' < 'B'

True

Capitalization matters, and capital letters are less than lowercase letters:

>>> 'a' != 'A'

True

>>> 'a' < 'A'

False

Every letter can be compared:

>>> ',' < '3'

True

We can compare a string and an integer for equality:

>>> 's' == 3

False

We can't compare values of two different types for ordering:

>>> 's' <= 3

Traceback (most recent call last):

File "<stdin>", line 1, in <module>>

TypeError: unorderable types: str() <= int()

### Testing For Substrings

The operator in checks whether a string appears anywhere inside another one (that is, whether a string is a substring of another).

>>> 'c' in 'aeiou'

False

>>> 'cad' in 'abracadabra'

True

>>> 'zoo' in 'ooze'

False

### String length: function len

The builtin function len returns the number of characters in a string:

>>> len('')

0

>>> len('abracadabra')

11

>>> len('Bwa' + 'ha' \* 10)

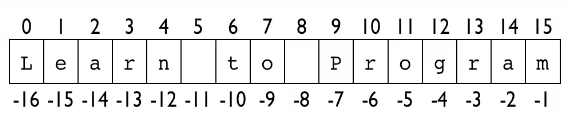
23

### Summary

|  |  |  |  |
| --- | --- | --- | --- |
| Description | Operator | Example | Result of example |
| equality | == | 'cat' == 'cat' | True |
| inequality | != | 'cat' != 'Cat' | True |
| less than | < | 'A' < 'a' | True |
| greater than | > | 'a' > 'A' | True |
| less than or equal | <= | 'a' <= 'a' | True |
| greater than or equal | >= | 'a' >= 'A' | True |
| contains | in | 'cad' in 'abracadabra' | True |
| length of str s | len(s) | len("abc") | 3 |

# str: indexing and slicing

### Indexing

An index is a position within the string. Positive indices count from the left-hand side with the first character at index 0, the second at index 1, and so on. Negative indices count from the right-hand side with the last character at index -1, the second last at index -2, and so on. For the string "Learn to Program", the indices are:  


Let s refer to 'Learn to Program'.

The first character of the string is at index 0 and can be accessed using this bracket notation:

>>> s[0]

'L'

>>> s[1]

'e'

Negative indices are used to count from the end (from the right-hand side):

>>> s[-1]

'm'

>>> s[-2]

'a'

### Slicing

We can extract more than one character using slicing. A slice is a substring from the start index up to but not including the end index. For example:

>>> s[0:5]

'Learn'

>>> s[6:8]

'to'

>>> s[9:16]

'Program'

More generally, the end of the string can be represented using its length:

>>> s[9:len(s)]

'Program'

The end index may be omitted entirely and the default is len(s):

>>> s[9:]

'Program'

Similarly, if the start index is omitted, the slice starts from index 0:

>>> s[:]

'Learn to Program'

>>> s[:8]

'Learn to'

Negative indices can be used for slicing too. The following three expressions are equivalent:

>>> s[1:8]

'earn to'

>>> s[1:-8]

'earn to'

>>> s[-15:-8]

'earn to'

### Modifying Strings

The slicing and indexing operations do not modify the string that they act on, so the string that s refers to is unchanged by the operations above. In fact, we cannot change a string. Operations like the following result in errors:

>>> s[6] = 'd'

Traceback (most recent call last):

File <"pyshell#19", line 1, in <module>

s[6] = 'd'

TypeError: 'str' object does not support item assignment

Imagine that we want to change string s to refer to 'Learned to Program'. The following expression evaluates to that 'Learned to Program': s[:5] + 'ed' + s[5:]

Variable s gets the new string: s = s[:5] + 'ed' + s[5:]

Notice that the string that s originally referred to was not modified: strings cannot be modified. Instead a new string was created and s was changed to point to that string.

# str Methods: Functions Inside Objects

### Methods

A method is a function inside of an object.

The general form of a method call is:  
  
object.method(arguments)

### String Methods

Consider the code:

>>> white\_rabbit = "I'm late! I'm late! For a very important date!"

To find out which methods are inside strings, use the function dir:

>>> dir(white\_rabbit)

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_',

'\_\_ge\_\_', '\_\_getattribute\_\_','\_\_getitem\_\_', '\_\_getnewargs\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_',

'\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mod\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_',

'\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_',

'\_\_subclasshook\_\_', 'capitalize', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',

'format', 'format\_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier',

'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',

'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',

'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',

'zfill']

Passing str as an argument to dir gets the same result:

>>> dir(str)

For many of the string methods, a new string is returned. Since strings are immutable, the original string is unchanged. For example, a lowercase version of the str that white\_rabbit refers to is returned when the method lower is called:

>>> white\_rabbit.lower()

>>> "i'm late! i'm late! for a very important date!"

>>> white\_rabbit

>>> "I'm late! I'm late! For a very important date!"

To get information about a method, such as the lower method, do the following:

>>> help(str.lower)

# for loop over str

### For Loops

The general form of a for loop over a string is:

for variable in str:

body

The variable refers to each character of the string in turn and executes the body of the loop for each character. For example:

>>> s = 'yesterday'

>>> for char in s:

... print(char)

...

y

e

s

t

e

r

d

a

y

### Accumulator pattern: numeric accumulator

Consider the code below, in which the variable num\_vowels is an accumulator:

def count\_vowels(s):

""" (str) -> int

Return the number of vowels in s. Do not treat letter y as a vowel

>>> count\_vowels('Happy Anniversary!')

5

>>> count\_vowels('xyz')

0

"""

num\_vowels = 0

for char in s:

if char in 'aeiouAEIOU':

num\_vowels = num\_vowels + 1

return num\_vowels

The loop in the function above will loop over each character that s refers to, in turn. The body of the loop is executed for each character, and when a character is a vowel, the if condition is True and the value that num\_vowels refers to is increased by one.

The variable num\_vowels is an accumulator, because it accumulates information. It starts out referring to the value 0 and by the end of the function it refers to the number of vowels in s.

### Accumulator pattern: string accumulator

In the following function, the variable vowels is also an accumulator:

def collect\_vowels(s):

""" (str) -> str

Return the vowels from s. Do not treat the letter

y as a vowel.

>>> collect\_vowels('Happy Anniversary!')

'aAiea'

>>> collect\_vowels('xyz')

''

"""

vowels = ''

for char in s:

if char in 'aeiouAEIOU':

vowels = vowels + char

return vowels

Variable vowels initially refers to the empty string, but over the course of the function it accumulates the vowels from s.

**IDLE's Debugger**

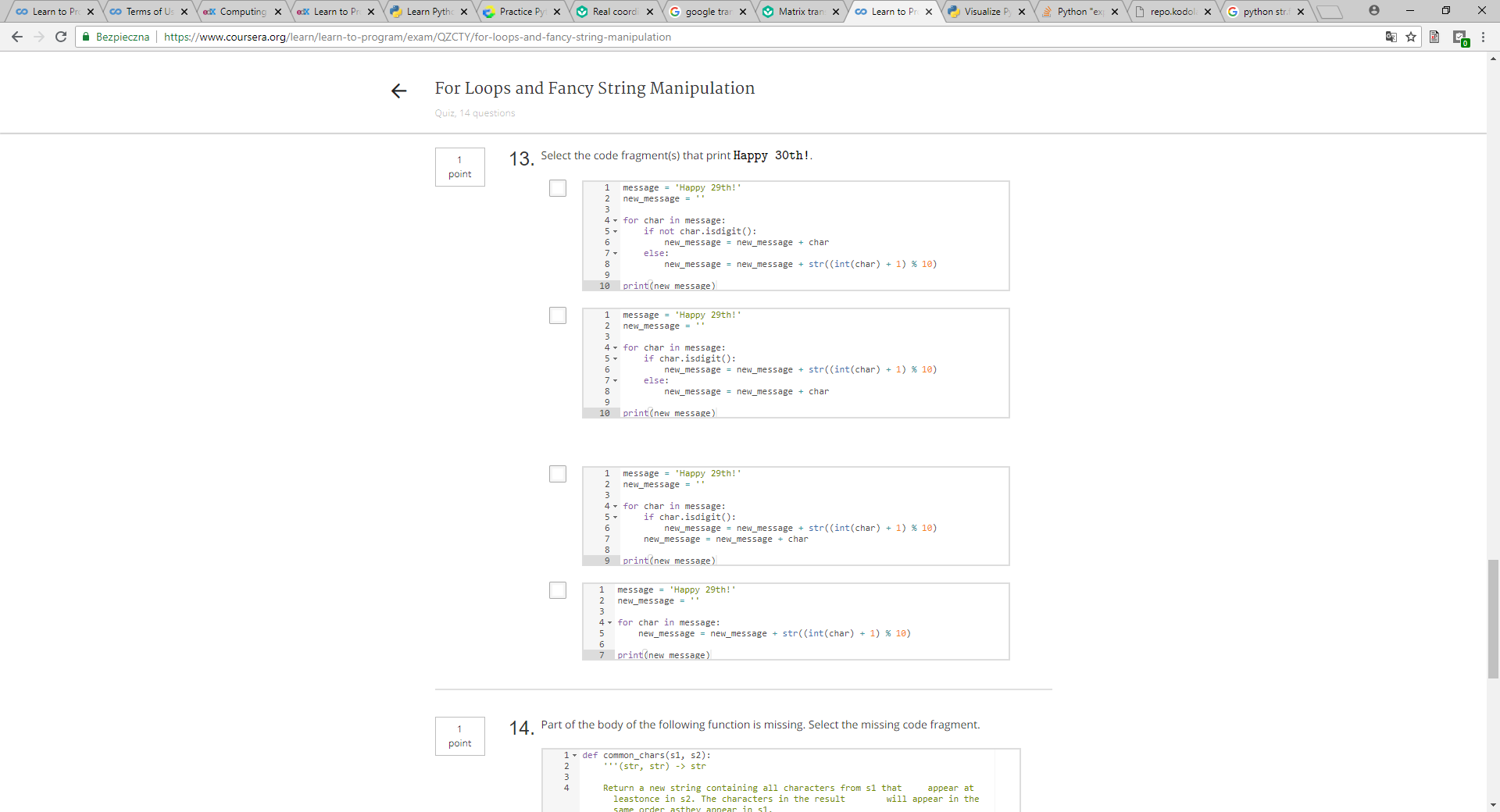
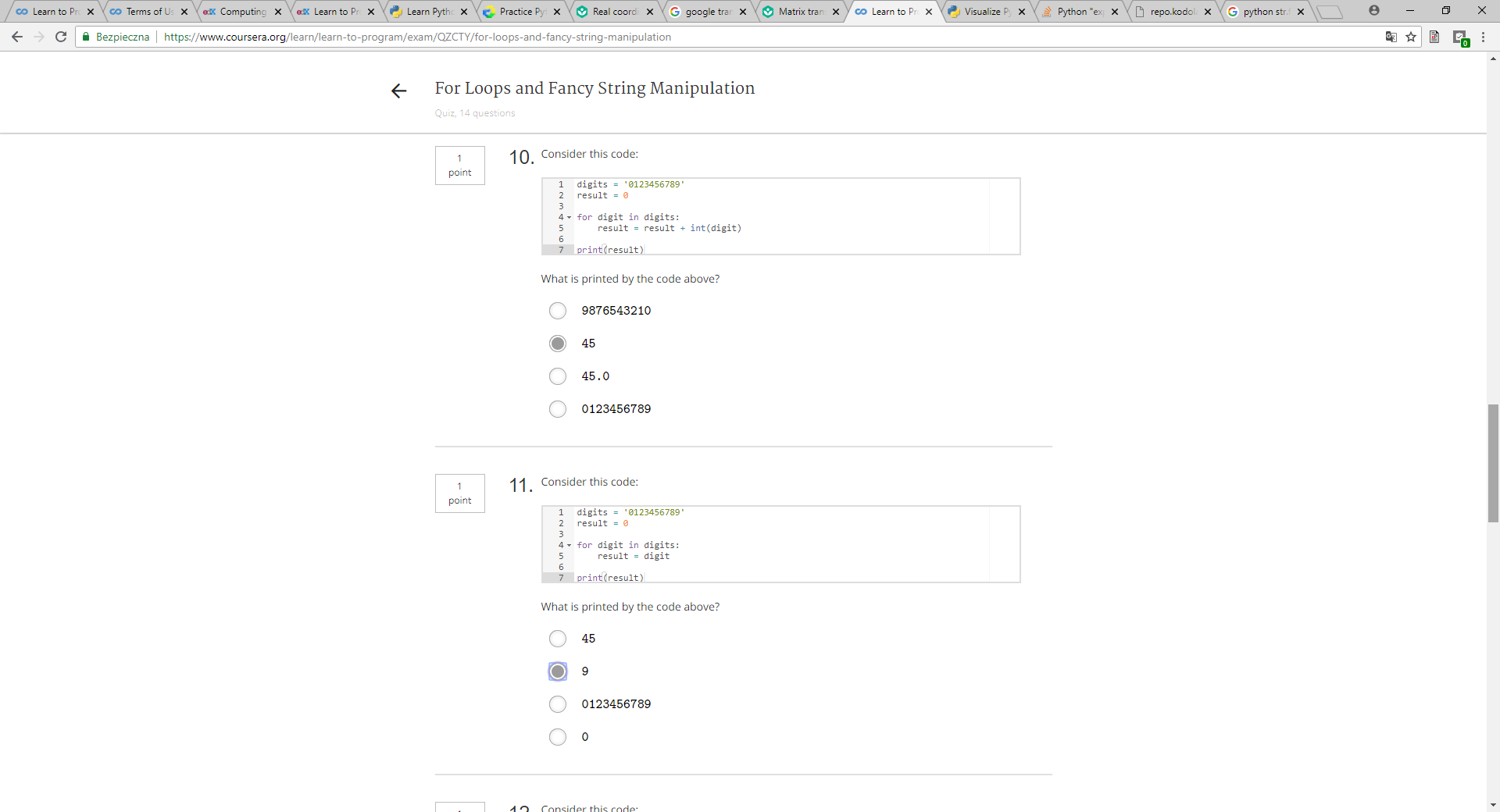
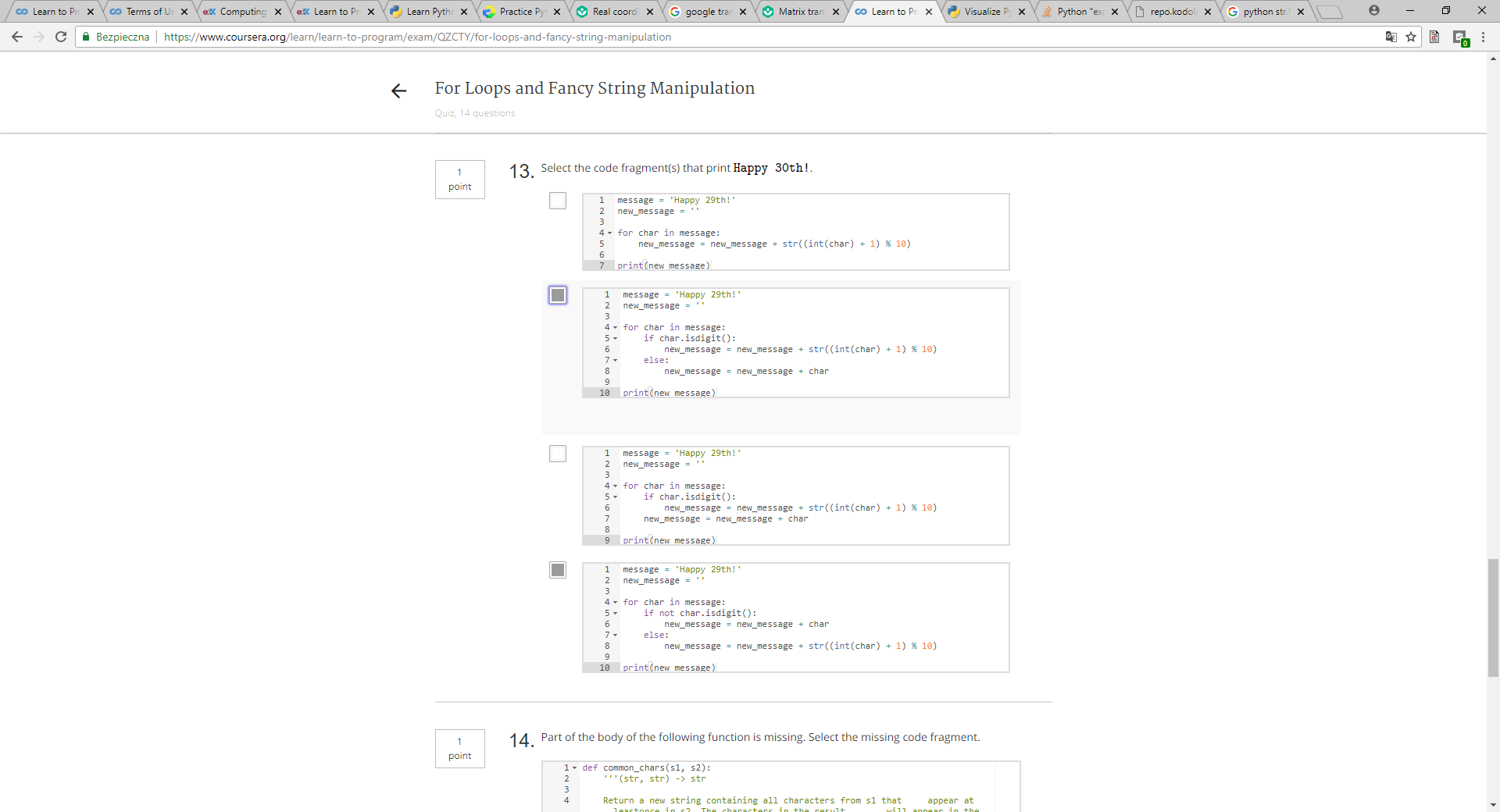
**Debug Control**

The Python Visualizer has limitations: it does not allow import statements, and it stops tracing after 300 steps. IDLE comes with a debugger, which is a tool that works a lot like the visualizer but without the pretty pictures. To run a program in IDLE's debugger, the steps are:

1. Make sure the Python Shell window is on top and select Debug->Debugger. This opens a window called "Debug Control".
2. Check the checkbox for Source.
3. Open the Python file where you have saved your program.
4. Select Run->Run Module. This will change the contents of Debug Control.

**Understanding The Debug Window**

Step is like Forward in the visualizer: it executes the current instruction. We click Step to "step" through the program. The Debug Control window will highlight the current line being executed. It will also show the current variables in the "Locals" pane. The middle pane shows the current stack frames and the current lines of code for each. We can switch back and forth between them to see the variables.

# while loops

The general form of a while loop:

while expression:

statements

The while condition, num < 100, is evaluated, and if it is True the statements in the loop body are executed. The loop condition is rechecked and if found to be True, the body executes again. This continues until the loop condition is checked and is False. For example:

>>> num = 2

>>> while num < 100:

num = num \* 2

print(num)

4

8

16

32

64

128

In the example above, there are 6 iterations: the loop body executes 6 times.

### Loops Conditions and Lazy Evaluation

**The problem**: print the characters of str s, up to the first vowel in s.

The first attempt at solving this problem works nicely when s contains one or more vowel, but results in an error if there are no vowels in s:

>>> i = 0

>>> s = 'xyz'

>>> while not (s[i] in 'aeiouAEIOU'):

print(s[i])

i = i + 1

x

y

z

Traceback (most recent call last):

File "<pyshell#73>", line 1, in <module>

while not (s[i] in 'aeiouAEIOU'):

IndexError: string index out of range

In the code above, the error occurs when s is indexed at i and i is outside of the range of valid indices. To prevent this error, add an additional condition is added to ensure that i is within the range of valid indices for s:

>>> i = 0

>>> s = 'xyz'

>>> while i < len(s) and not (s[i] in 'aeiouAEIOU'):

print(s[i])

i = i + 1

x

y

z

Because Python evaluates the and using lazy evaluation, if the first operand is False, then the expression evaluates to False and the second operand is not even evaluated. That prevents the IndexError from occurring.

def get\_answer(prompt):

''' (str) -> str

Use prompt to ask the user for a "yes" or "no"

answer and continue asking until the user gives

a valid response. Return the answer.

'''

answer = input(prompt)

while not (answer == 'yes' or answer == 'no'):

answer = input(prompt)

return answer

def up\_to\_vowel(s):

''' (str) -> str

Return a substring of s from index 0 up to but

not including the first vowel in s.

>>> up\_to\_vowel('hello')

'h'

>>> up\_to\_vowel('there')

'th'

>>> up\_to\_vowel('cs')

'cs'

'''

before\_vowel = ''

i = 0

while i < len(s) and not (s[i] in 'aeiouAEIOU'):

before\_vowel = before\_vowel + s[i]

i = i + 1

return before\_vowel

# Comments

### The Why and How of Comments

As your programs get longer and more complicated, some additional English explanation can be used to help you and other programmers read your code. These explanations called comments document your code, much the way docstrings document your functions.

A comment begins with the number sign character (#) and goes until the end of the line. One name for this character is the hash character. Python ignores any lines that start with this character.

Comments are intended for programmers to read, and are usually there to explain the purpose of a function, as well as to describe relationships between your variables. Comments are to help you, and anyone else who is reading/using your code, to remember or understand the purpose of a given variable or function in a program.

**Type list**

**Overview**

Our programs will often work with collections of data. One way to store these collections of data is using Python's type list.

The general form of a list is:

[*expr1*, *expr2*, ..., *exprN*]

For example, here is a list of three grades:

grades = [80, 90, 70]

**List Operations**

Like strings, lists can be indexed:

>>> grades[0]

80

>>> grades[1]

90

>>> grades[2]

70

Lists can also be sliced, using the same notation as for strings:

>>> grades[0:2]

[80, 90]

The in operator can also be applied to check whether a value is an item in a list.

>>> 90 in grades

True

>>> 60 in grades

False

Several of Python's built-in functions can be applied to lists, including:

* len(list): return the length of list.
* min(list): return the smallest element in list.
* max(list): return the largest element in list.
* sum(list): return the sum of elements of list (where list items must be numeric).

For example, here are some calls to those built-in functions:

>>> len(grades)

3

>>> min(grades)

70

>>> max(grades)

90

>>> sum(grades)

240

**Types of list elements**

Lists elements may be of any type. For example, here is a list of str:

subjects = ['bio', 'cs', 'math', 'history']

Lists can also contain elements of more than one type. For example, a street address can be represented by a list of [int, str]:

street\_address = [10, 'Main Street']

**for loops over list**

Similar to looping over the characters of a string, it is possible to iterate over the elements of a list. For example:

>>> for grade in grades:

print(grade)

80

90

70

The general form of a for loop over a list is:

for *variable* in *list*:

*body*

Python 3.2.3 (v3.2.3:3d0686d90f55, Apr 10 2012, 11:25:50)

[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin

Type "copyright", "credits" or "license()" for more information.

>>> grades = [80, 90, 70]

>>> grades[0]

80

>>> grades[1]

90

>>> grades[2]

70

>>> grades[1:2]

[90]

>>> grades[0:2]

[80, 90]

>>> 90 in grades

True

>>> 60 in grades

False

>>> len(grades)

3

>>> min(grades)

70

>>> max(grades)

90

>>> sum(grades)

240

>>> subjects = ['bio', 'cs', 'math', 'history']

>>> len(subjects)

4

>>> min(subjects)

'bio'

>>> max(subjects)

'math'

>>> sum(subjects)

Traceback (most recent call last):

File "<pyshell#16>", line 1, in <module>

sum(subjects)

TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> street\_address = [10, 'Main Street']

>>> for grade in grades:

print(grade)

80

90

70

>>> for item in subjects:

print(item)

bio

cs

math

history

# list Methods

### Methods

A method is a function inside an object. You can find out the methods in type list by typing dir(list).

### Modifying Lists

The table below contains methods that modify lists.

|  |  |  |
| --- | --- | --- |
| Method | Description | Example |
| list.append(object) | Append object to the end of list. | >>> colours = ['yellow', 'blue']  >>> colours.append('red')  >>> print(colours)  ['yellow', 'blue', 'red'] |
| list.extend(list) | Append the items in the list parameter to the list. | >>> colours.extend(['pink', 'green'])  >>> print(colours)  ['yellow', 'blue', 'red', 'pink', 'green'] |
| list.pop([index]) | Remove the item at the end of the list; optional index to remove from anywhere. | >>> colours.pop()  'green'  >>> print(colours)  ['yellow', 'blue', 'red', 'pink']  >>> colours.pop(2)  'red'  >>> print(colours)  ['yellow', 'blue', 'pink'] |
| list.remove(object) | Remove the first occurrence of the object; error if not there. | >>> colours.remove('green')  Traceback (most recent call last):  File "<pyshell#10>", line 1, in <module>  colours.remove('green')  ValueError: list.remove(x): x not in list  >>> colours.remove('pink')  >>> print(colours)  ['yellow', 'blue'] |
| list.reverse() | Reverse the list. | >>> grades = [95, 65, 75, 85]  >>> grades.reverse()  >>> print(grades)  [85, 75, 65, 95] |
| list.sort() | Sort the list from smallest to largest. | >>> grades.sort()  >>> print(grades)  [65, 75, 85, 95] |
| list.insert(int, object) | Insert object at the given index, moving items to make room. | >>> grades.insert(2, 80)  >>> print(grades)  [65, 75, 80, 85, 95] |

### Getting Information from Lists

The table below contains methods that return information about lists.

|  |  |  |
| --- | --- | --- |
| Method | Description | Example |
| list.count(object) | Return the number of times object occurs in list. | >>> letters = ['a', 'a', 'b', 'c']  >>> letters.count('a')  2 |
| list.index(object) | Return the index of the first occurrence of object; error if not there. | >>> letters.index('a')  0  >>> letters.index('d')  Traceback (most recent call last):  File "<pyshell#24>", line 1, in <module>  letters.index('d')  ValueError: 'd' is not in list |
|  |  |  |

Python 3.2.3 (v3.2.3:3d0686d90f55, Apr 10 2012, 11:25:50)

[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin

Type "copyright", "credits" or "license()" for more information.

>>> dir(list)

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_delitem\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_iadd\_\_', '\_\_imul\_\_', '\_\_init\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_reversed\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_setitem\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

>>> colours = []

>>> prompt = 'Enter another one of your favourite colours (type return to end): '

>>>

>>> colour = input(prompt)

Enter another one of your favourite colours (type return to end): blue

>>> colour

'blue'

>>> colours

[]

>>> while colour != '':

colours.append(colour)

colour = input(prompt)

Enter another one of your favourite colours (type return to end): yellow

Enter another one of your favourite colours (type return to end): brown

Enter another one of your favourite colours (type return to end):

>>> colours

['blue', 'yellow', 'brown']

>>> colours.extend(['hot pink', 'neon green'])

>>> colours

['blue', 'yellow', 'brown', 'hot pink', 'neon green']

>>> colours.pop()

'neon green'

>>> colours

['blue', 'yellow', 'brown', 'hot pink']

>>> colours.pop(2)

'brown'

>>> colours

['blue', 'yellow', 'hot pink']

>>> colours.remove('black')

Traceback (most recent call last):

File "<pyshell#18>", line 1, in <module>

colours.remove('black')

ValueError: list.remove(x): x not in list

>>> if colours.count('yellow') > 0:

colours.remove('yellow')

>>> colours

['blue', 'hot pink']

>>> if 'yellow' in colours:

colours.remove('yellow')

>>> colours

['blue', 'hot pink']

>>> colours.extend('auburn', 'taupe', 'magenta')

Traceback (most recent call last):

File "<pyshell#27>", line 1, in <module>

colours.extend('auburn', 'taupe', 'magenta')

TypeError: extend() takes exactly one argument (3 given)

>>> colours.extend(['auburn', 'taupe', 'magenta'])

>>> colours

['blue', 'hot pink', 'auburn', 'taupe', 'magenta']

>>> colours.sort()

>>> colours

['auburn', 'blue', 'hot pink', 'magenta', 'taupe']

>>> colours.reverse()

>>> colours

['taupe', 'magenta', 'hot pink', 'blue', 'auburn']

>>> colours.insert(-2, 'brown')

>>> colours

['taupe', 'magenta', 'hot pink', 'brown', 'blue', 'auburn']

>>> colours.index('neon green')

Traceback (most recent call last):

File "<pyshell#36>", line 1, in <module>

colours.index('neon green')

ValueError: 'neon green' is not in list

>>> if 'hot pink' in colours:

where = colours.index('hot pink')

colours.pop(where)

'hot pink'

>>> colours

['taupe', 'magenta', 'brown', 'blue', 'auburn']

>>>

# Mutability and Aliasing

### Mutability

We say that lists are *mutable*: they can be modified. All the other types we have seen so far (str, int, float and bool) are *immutable*: they cannot be modified.

Here are several examples of lists being modified:

>>> classes = ['chem', 'bio', 'cs', 'eng']

>>>

>>> # Elements can be added:

>>> classes.append('math')

>>> classes

['chem', 'bio', 'cs', 'eng', 'math']

>>>

>>> # Elements can be replaced:

>>> classes[1] = 'soc'

>>> classes

['chem', 'soc', 'cs', 'eng', 'math']

>>>

>>> # Elements can be removed:

>>> classes.pop()

'math'

>>> classes

['chem', 'soc', 'cs', 'eng']

### Aliasing

Consider the following code:

>>> lst1 = [11, 12, 13, 14, 15, 16, 17]

>>> lst2 = lst1

>>> lst1[-1] = 18

>>> lst2

[11, 12, 13, 14, 15, 16, 18]

After the second statement executes, lst1 and lst2 both refer to the same list. When two variables refer to the same objects, they are aliases. If that list is modified, both of lst1 and lst2 will see the change.

Wyświetlanie liczb z zakresu:  
>>> for num in range(10):  
 print(num)

0

1

2

3

4

5

6

7

8

9

>>> for num in range(2,25,3): //który wyraz, zakres max 25, czyli 24, co ile skok czyli 3

print(num)

2

5

8

11

14

17

20

23

>>>

# Range - zakres

### The Built-in Function: range

Python has a built-in function called range that is useful to use when you want to generate a sequence of numbers. You can type help(range) in IDLE if you ever need a reminder.

The example below will print the integers 0 to 9, inclusive.

for i in range(10):

print (i)

The form of range is:

range([start,] stop[, step]):

return a virtual sequence of numbers from start to stop by step

### Applications of Range

There are other options you can specify to range. One option is to let range generate the numbers corresponding to indices of a string or a list.

s = 'computer science'

for i in range(len(s)):

print(i)

You can also tell range what index to start at. For instance, the example below starts at index 1 (as opposed to the default which is 0).

for i in range(1, len(s)):

print(i)

You can even specify the "step" for range. The default stepping size is 1, which means that numbers increment by 1. The example below starts at index 1 and its step size is there (goes to every third index).

for i in range(1, len(s), 3):

print(i)

# for loops over indices

### range

Here is the top part of the help for range:

class range(object)

| range([start,] stop[, step]) -> range object

|

| Returns a virtual sequence of numbers from start to stop by step.

The stop value is not included.

range is typically used in a for loop to iterate over a sequence of numbers. Here are some examples:

# Iterate over the numbers 0, 1, 2, 3, and 4.

for i in range(5):

# Iterate over the numbers 2, 3, and 4.

for i in range(2, 5):

# Iterate over the numbers 3, 6, 9, 12, 15, and 18.

for i in range(3, 20, 3):

### Iterating over the indices of a list

Because len returns the number of items in a list, it can be used with range to iterate over all the indices. This loop prints all the values in a list:

for i in range(len(lst)):

print(lst[i])

This also gives us flexibility to process only part of a list. For example, We can print only the first half of the list:

for i in range(len(lst) // 2):

print(lst[i])

Or every other element:

for i in range(0, len(lst), 2):

print(lst[i])

### Not "What" but "Where"

Previously, we have written loops over characters in a string or items in a list. However, sometimes there are problems where knowing the value of the items in a list or the characters in a string is not enough; we need to know where it occurs (i.e. its index).

### Example 1

The first example is described below:

def count\_adjacent\_repeats(s):

''' (str) -> int

Return the number of occurrences of a character and an adjacent character

being the same.

>>> count\_adjacent\_repeats('abccdeffggh')

3

'''

repeats = 0

for i in range(len(s) - 1):

if s[i] == s[i + 1]:

repeats = repeats + 1

return repeats

We want to compare a character in the string with another character in the string beside it. This is why we iterate over the indices because the location is important, and only knowing the value of the character does not provide us with enough information. This is how we are able to count repeated characters in a string. We can't execute the body of the loop if i is len(s) - 1 because we compare to s[i + 1], and that would produce an IndexError.

### Example 2

The second example is described below:

def shift\_left(L):

''' (list) -> NoneType

Shift each item in L one position to the left and shift the first item to

the last position.

Precondition: len(L) >= 1

>>> shift\_left(['a', 'b', 'c', 'd'])

'''

first\_item = L[0]

for i in range(len(L) - 1):

L[i] = L[i + 1]

L[-1] = first\_item

For the same reasons as above, merely knowing the value of the items in the list is not enough since we need to know where the items are located; we need to know the index (position) of the item in the list.

def shift\_left(L):

''' (list) -> NoneType

Shift each item in L one position to the left

and shift the first item to the last position.

Precondition: len(L) >= 1

'''

first\_item = L[0]

for i in range(1, len(L)):

L[i - 1] = L[i]

L[-1] = first\_item

def count\_adjacent\_repeats(s):

''' (str) -> int

Return the number of occurrences of a character and

an adjacent character being the same.

>>> count\_adjacent\_repeats('abccdeffggh')

3

'''

repeats = 0

for i in range(len(s) - 1):

if s[i] == s[i + 1]:

repeats = repeats + 1

return repeats

# Parallel Lists and Strings

### Correspondings Elements

Two lists are parallel if they have the same length and the items at each index are somehow related. The items at the same index are said to be at corresponding positions.

Consider these two lists:

list1 = [1, 2, 3]

list2 = [2, 4, 2]

In these two lists, the corresponding element of list1[0] is list2[0], the corresponding element of list2[1] is list1[1], and so on.

### Example of Corresponding Elements

def match\_characters(s1, s2):

''' (str, str) -> int

Return the number of characters in s1 that are the same as the character

at the corresponding position of s2.

Precondition: len(s1) == len(s2)

>>> match\_characters('ate', 'ape')

2

>>> match\_characters('head', 'hard')

2

'''

num\_matches = 0

for i in range(len(s1)):

if s1[i] == s2[i]:

num\_matches = num\_matches + 1

return num\_matches

The function above counts the corresponding elements of the two strings that are the same character. If a character of s1 at index i is the same as the character of s2 at the same index, then we increment num\_matches by 1 (since they match). Otherwise, we continue on to the next pair of corresponding elements and compare them.

def count\_matches(s1, s2):

''' (str, str) -> int

Return the number of positions in s1 that contain the

same character at the corresponding position of s2.

Precondition: len(s1) == len(s2)

>>> count\_matches('ate', 'ape')

2

>>> count\_matches('head', 'hard')

2

'''

num\_matches = 0

for i in range(len(s1)):

if s1[i] == s2[i]:

num\_matches = num\_matches + 1

return num\_matches

def sum\_items(list1, list2):

''' (list of number, list of number) -> list of number

Return a new list in which each item is the sum of the items at the

corresponding position of list1 and list2.

Precondition: len(list1) == len(list2)

>> sum\_items([1, 2, 3], [2, 4, 2])

[3, 6, 5]

'''

sum\_list = []

for i in range(len(list1)):

sum\_list.append(list1[i] + list2[i])

return sum\_list

# Nested Lists

Lists can contain items of any type, including other lists. These are called nested lists.

Here is an example.

>>> grades = [['Assignment 1', 80], ['Assignment 2', 90], ['Assignment 3', 70]]

>>> grades[0]

['Assignment 1', 80]

>>> grades[1]

['Assignment 2', 90]

>>> grades[2]

['Assignment 3', 70]

To access a nested item, first select the sublist, and then treat the result as a regular list.

For example, to access 'Assignment 1', we can first get the sublist and then use it as we would a regular list:

>>> sublist = grades[0]

>>> sublist

['Assignment 1', 80]

>>> sublist[0]

'Assignment 1'

>>> sublist[1]

80

Both sublist and grades[0] contain the memory address of the ['Assignment 1', 80] nested list.

We can access the items inside the nested lists like this:

>>> grades[0][0]

'Assignment 1'

>>> grades[0][1]

80

>>> grades[1][0]

'Assignment 2'

>>> grades[1][1]

90

>>> grades[2][0]

'Assignment 3'

>>> grades[2][1]

70

# Nested Loops

### Bodies of Loops

The bodies of loops can contain any statements, including other loops. When this occurs, this is known as a nested loop.

Here is a nested loop involving 2 for loops:

for i in range(10, 13):

for j in range(1, 5):

print(i, j)

Here is the output:

10 1

10 2

10 3

10 4

11 1

11 2

11 3

11 4

12 1

12 2

12 3

12 4

Notice that when i is 10, the inner loop executes in its entirety, and only after j has ranged from 1 through 4 is i assigned the value 11.

### Example of Nested Loops

def calculate\_averages(grades):

''' (list of list of number) -> list of float

Return a new list in which each item is the average of the grades in the

inner list at the corresponding position of grades.

>>> calculate\_averages([[70, 75, 80], [70, 80, 90, 100], [80, 100]])

[75.0, 85.0, 90.0]

'''

averages = []

# Calculate the average of each sublist and append it to averages.

for grades\_list in grades:

# Calculate the average of grades\_list.

total = 0

for mark in grades\_list:

total = total + mark

averages.append(total / len(grades\_list))

return averages

In calculate\_averages, the outer for loop iterates through each sublist in grades. We then calculate the average of that sublist using a nested, or inner, loop, and add the average to the accumulator (the new list, averages).

def averages(grades):

'''

(list of list of number) -> list of float

Return a new list in which each item is the average of the

grades in the inner list at the corresponding position of

grades.

>>> averages([[70, 75, 80], [70, 80, 90, 100], [80, 100]])

[75.0, 85.0, 90.0]

'''

averages = []

for grades\_list in grades:

# Calculate the average of grades\_list and append it

# to averages.

total = 0

for mark in grades\_list:

total = total + mark

averages.append(total / len(grades\_list))

return averages

# Reading Files

Information stored in files can be accessed by a Python program. To get access to the contents of a file, you need to open the file in your program. When you are done using a file, you should close it.

### Opening and Closing A File

Python has a built-in function open that can open a file for reading.

The form of open is open(filename, mode), where mode is 'r' (to open for reading), 'w' (to open for writing), or 'a' (to open for appending to what is already in the file).

This opens a file called In Flanders Fields.txt for reading:

flanders\_file = open('In Flanders Fields.txt', 'r')

Note that if the file is saved in the same directory as your program, you can simply write the name of the file, as what was done in the above example. However, if it is not saved in the same directory, you must provide the path to it.

To close a file, you write flanders\_file.close() .

There are four standard ways to read from a file. Some use these methods:

readline(): read and return the next line from the file, including the newline character (if it exists). Return the empty string if there are no more lines in the file.

readlines(): read and return all lines in a file in a list. The lines include the newline character.

read(): read the whole file as a single string.

|  |  |  |
| --- | --- | --- |
| **Approach** | **Code** | **When to use it** |
| The readline approach | file = open(filename, 'r')  # Read lines until we reach the  # place in the file that we want.  line = file.readline()  while we are not at the place we want:  line = file.readline()  # Now we have reached the section  # of the file we want to process.  line = file.readline()  while we are not at the end of the section:  process the line  line = file.readline()  flanders\_file.close() | When you want to process only part of a file. |
| The for line in file approach | file = open(filename, 'r')  for line in file:  process the line  file.close() | When you want to process every line in the file one at a time. |
| The read approach | file = open(filename, 'r')  contents = file.read()  now process contents  file.close() | When you want to read the whole file at once and use it as a single string. |
| The readlines approach | file = open(filename, 'r')  # Get the contents as a list of strings.  contents\_list = file.readlines()  process contents\_list using indexing to  access particular lines from the file  file.close() | When you want to examine each line of a file by index. |

### Examples from the video

Here are the code examples that appeared in the video. All of them read the entire file into a string and print that string.

#### The readline approach

flanders\_file = open(flanders\_filename, 'r')

flanders\_poem = ''

line = flanders\_file.readline()

while line != "":

flanders\_poem = flanders\_poem + line

line = flanders\_file.readline()

print(flanders\_poem)

flanders\_file.close()

#### The for line in file approach

flanders\_file = open(flanders\_filename, 'r')

flanders\_poem = ''

for line in flanders\_file:

flanders\_poem = flanders\_poem + line

print(flanders\_poem)

flanders\_file.close()

#### The read approach

flanders\_file = open(flanders\_filename, 'r')

flanders\_poem = flanders\_file.read()

print(flanders\_poem)

flanders\_file.close()

#### The readlines approach

flanders\_file = open(flanders\_filename, 'r')

flanders\_poem = ''

flanders\_list = flanders\_file.readlines()

for line in flanders\_list:

flanders\_poem = flanders\_poem + line

print(flanders\_poem)

flanders\_file.close()

# Write Files

### Writing To A File Within A Python Program

In order to write to a file, we use file.write(str). This method writes a string to a file. Method write works like Python's print function, except that it does not add a newline character.

### File dialogs

Module tkinter has a submodule called filedialog. We import it like this:

import tkinter.filedialog

Function askopenfilename asks the user to select a file to open:

tkinter.filedialog.askopenfilename()

This function returns the full path to the file, so we can use that when we call function open to open that file.

from\_filename = tkinter.filedialog.askopenfilename()

Function asksaveasfilename asks the user to select a file to save to, and provides a warning if the file already exists.

to\_filename = tkinter.filedialog.asksaveasfilename()

### Example

### Below is a program that copies a file, but puts "Copy" as the first line of the copied file.

In order to prompt a user for a file.

Now we can open the file we want to read from and get the contents:

from\_file = open(from\_filename, 'r')

contents = from\_file.read()

from\_file.close()

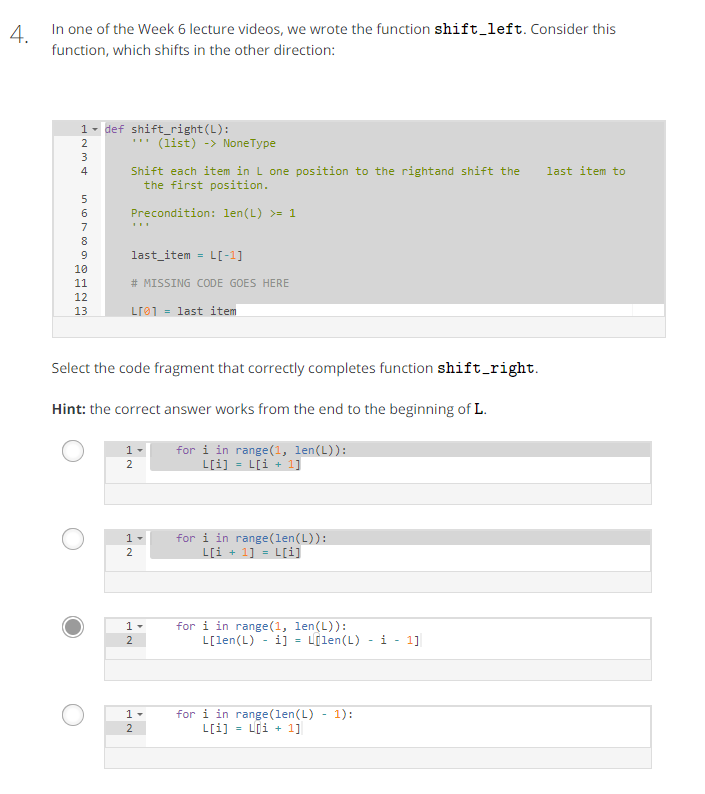
And we can open the file we want to write to and write the contents:

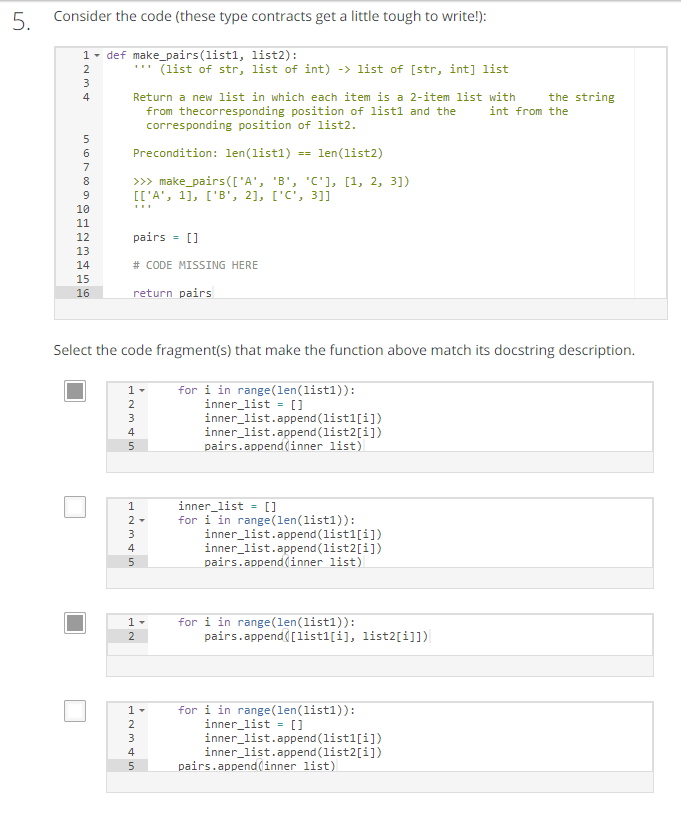
to\_file = open(to\_filename, 'w')

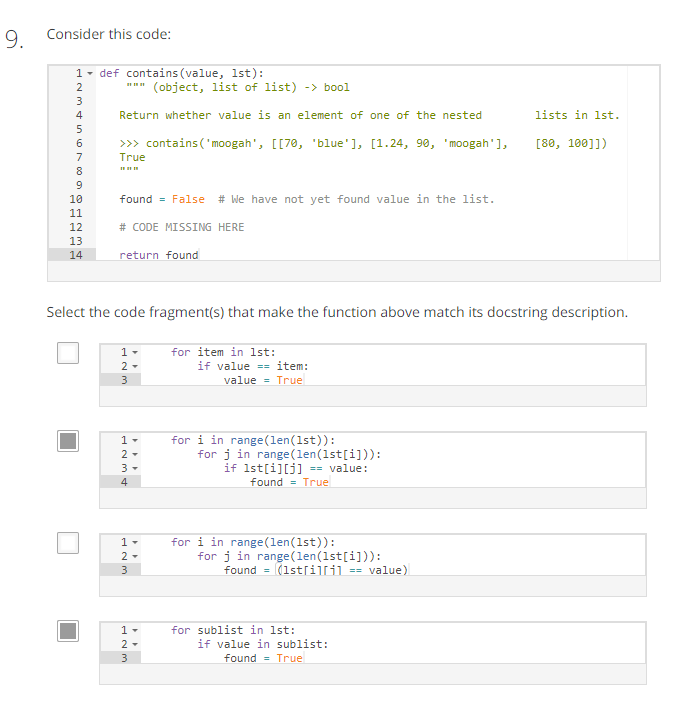
to\_file.write('Copy\n') # We have to add the newline ourselves.

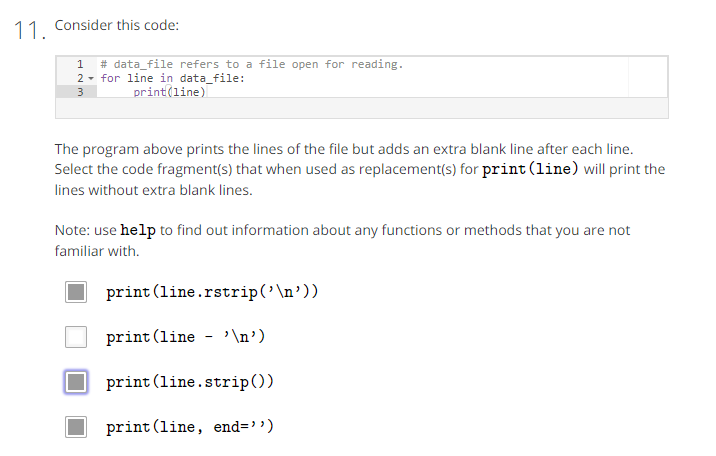
to\_file.write(contents) # Now write the contents of the file.

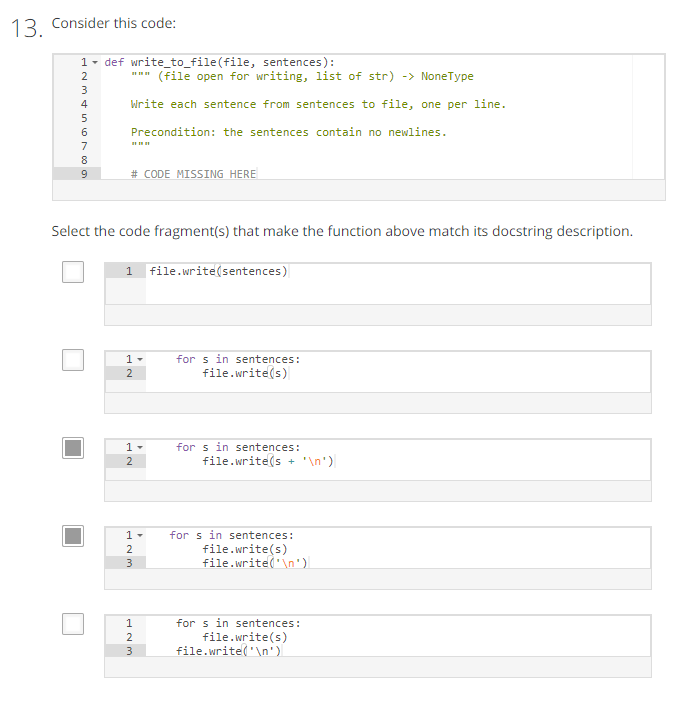
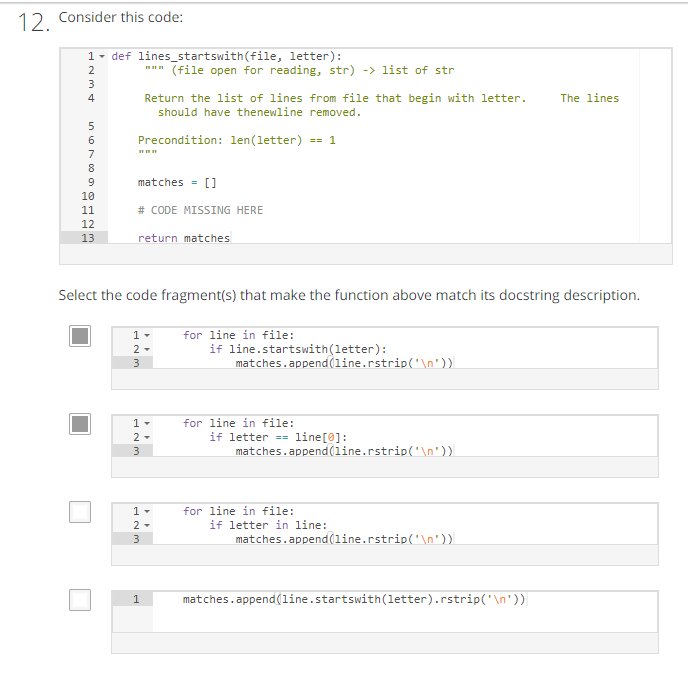
to\_file.close()

4 dobrze 

5 dobrze

9 dobrze



12 dobrze 13 dobrze

Napis – pobranie słów z tekstu:

>>> s = 'ala ma kota a kot ma ale'

>>> s

'ala ma kota a kot ma ale'

>>> s.split(' ')

['ala', 'ma', 'kota', 'a', 'kot', 'ma', 'ale']

>>>

# Tuples

### Immutable Sequences

Tuples are immutable sequences: they cannot be modified. Tuples and lists have much in common, but lists are mutable sequences: they can be modified.

Tuples use parentheses instead of square brackets:

lst = ['a', 3, -0.2]

tup = ('a', 3, -0.2)

Once created, items in lists and tuples are accessed using the same notation:

>>> lst[0]

'a'

>>> tup[0]

'a'

Slicing can be used with both:

>>> lst[:2]

['a', 3]

>>> tup[:2]

('a', 3)

Tuples cannot be modified:

>>> tup[0] = 'b'

TypeError: 'tuple' object does not support item assignment

Tuples have fewer methods than lists. In fact, the only regular methods are count and index:

>>> dir(list)

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_delitem\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_',

'\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_iadd\_\_', '\_\_imul\_\_', '\_\_init\_\_',

'\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_',

'\_\_repr\_\_', '\_\_reversed\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_setitem\_\_', '\_\_sizeof\_\_', '\_\_str\_\_',

'\_\_subclasshook\_\_', '**append', 'count', extend', 'index', 'insert', 'pop', 'remove', 'reverse', sort'**]

>>> dir(tuple)

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_',

'\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getnewargs\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_iter\_\_',

'\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_',

'\_\_repr\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', **'count', 'index'**]

The rest of the list methods are not available for tuple because they modify the object, and tuples, being immutable, cannot be modified.

A for can be used to iterate over the values in a tuple:

>>> tup = ('a', 3, -0.2)

>>> for item in tup:

print(item)

a

3

-0.2

A tuple can be passed as an argument to the built-in function len:

>>> len(tup)

3

It is also possible to iterate over the indices of a tuple:

>>> for i in range(len(tup)):

print(tup[i])

a

3

-0.2

## + info z Wikipedii **Krotki[**[**edytuj**](https://pl.wikibooks.org/w/index.php?title=Zanurkuj_w_Pythonie/Krotki&action=edit&section=1)**]**

Krotka (ang. *tuple*) jest niezmienną listą. Zawartość krotki określamy tylko podczas jej tworzenia. Potem nie możemy już jej zmienić.

**Przykład. *Definiowanie krotki***

>>> t = ("a", "b", "mpilgrim", "z", "element") #(1)

>>> t

('a', 'b', 'mpilgrim', 'z', 'element')

>>> t[0] #(2)

'a'

>>> t[-1] #(3)

'element'

>>> t[1:3] #(4)

('b', 'mpilgrim')

# Type dict

### Dictionary

Another way to store collections of data is using Python's dictionary type: dict.

The general form of a dictionary is:

{key1: value1, key2: value2, ..., keyN: valueN}

Keys must be unique. Values may be duplicated. For example:

asn\_to\_grade = {'A1': 80, 'A2': 90, 'A3': 90}

In the example above, the keys are unique: 'A1', 'A2' and 'A3'. The values are not unique: 80, 90 and 90.

### How To Modify Dictionaries

Dictionaries are mutable: they can be modified. There are a series of operations and methods you can apply to dictionaries which are outlined below.

|  |  |  |
| --- | --- | --- |
| Operation | Description | Example |
| object in dict | Checks whether object is a key in dict. | >>> asn\_to\_grade = {'A1': 80, 'A2': 90, 'A3': 90}  >>> 'A1' in asn\_to\_grade  True  >>> 80 in asn\_to\_grade  False |
| len(dict) | Returns the number of keys in dict. | >>> asn\_to\_grade = {'A1': 80, 'A2': 90, 'A3': 90}  >>> len(asn\_to\_grade)  3 |
| del dict[key] | Removes a key and its associated value from dict. | >>> asn\_to\_grade = {'A1': 80, 'A2': 90, 'A3': 90}  >>> del asn\_to\_grade['A1']  >>> asn\_to\_grade  {'A3': 90, 'A2': 90} |
| dict[key] = value | If key does not exist in dict, adds key and its associated value to dict. If key exists in dict, updates dict by setting the value associated with key to value. | >>> asn\_to\_grade = {'A1' : 80, 'A2': 90, 'A3' : 90}  >>> asn\_to\_grade['A4'] = 70  >>> asn\_to\_grade  {'A1': 80, 'A3': 90, 'A2': 90, 'A4': 70} |

### Accessing Information From Dictionaries

Dictionaries are unordered. That is, the order the key-value pairs are added to the dictionary has no effect on the order in which they are accessed. For example:

>>> asn\_to\_grade = {'A1': 80, 'A2': 70, 'A3': 90}

>>> for assignment in asn\_to\_grade:

print(assignment)

A1

A3

A2

The for-loop above printed out the keys of the dictionary. It is also possible to print out the values:

>>> asn\_to\_grade = {'A1': 80, 'A2': 70, 'A3': 90}

>>> for assignment in asn\_to\_grade:

print(asn\_to\_grade[assignment])

80

90

70

Finally, both the keys are values can be printed:

>>> asn\_to\_grade = {'A1': 80, 'A2': 70, 'A3': 90}

>>> for assignment in asn\_to\_grade:

print(assignment, asn\_to\_grade[assignment])

A1 80

A3 90

A2 70

### Empty Dictionaries

A dictionary can be empty. For example:

d = {}

### Heterogeneous Dictionaries

A dictionary can have keys of different types. For example, one key can be of type int and another of type str:

d = {'apple': 1, 3: 4}

### Immutable Keys

The keys of a dictionary must be immutable. Therefore, lists, dictionary and other mutable types cannot be used as keys. The following results in an error:

d[[1, 2]] = 'banana'

Since lists are mutable, they cannot be keys. Instead, to use a sequence as a key, type tuple can be used:

d[(1, 2)] = 'banana'

**Inverting a Dictionary**

**Switching Keys and Values**

Dictionaries have keys that are unique and each key has a value associated with it. For example, here is a dictionary mapping fruit to their colours:

fruit\_to\_colour = {'watermelon': 'green', 'pomegranate': 'red',

'peach': 'orange', 'cherry': 'red', 'pear': 'green',

'banana': 'yellow', 'plum': 'purple', 'orange': 'orange'}

To invert the dictionary, that is, switch the mapping to be colours to fruit, here is one approach:

>>> colour\_to\_fruit = {}

>>> for fruit in fruit\_to\_colour:

colour = fruit\_to\_colour[fruit]

colour\_to\_fruit[colour] = fruit

>>> colour\_to\_fruit

{'orange': 'orange', 'purple': 'plum', 'green': 'pear', 'yellow': 'banana', 'red': 'pomegranate'}

The resulting dictionary is missing some fruit. This happens since colours, which are keys, are unique so later assignments using the same colour replace earlier entries. A way to remedy this is to map colours to a list of fruit.

**Mapping A Key To A List**

For the example above, we need to consider two cases when adding a colour and a fruit to the dictionary:

1. If the colour is not a key in the dictionary, add it with its value being a single element a list consisting of the fruit.
2. If the colour is already a key, append the fruit to the list of fruit associated with that key.

>>> colour\_to\_fruit = {}

>>> for fruit in fruit\_to\_colour:

# What colour is the fruit?

colour = fruit\_to\_colour[fruit]

if not (colour in colour\_to\_fruit):

colour\_to\_fruit[colour] = [fruit]

else:

colour\_to\_fruit[colour].append(fruit)

>>> colour\_to\_fruit

{'orange': ['peach', 'orange'], 'purple': ['plum'], 'green': ['watermelon', 'pear'], 'yellow': ['banana'], 'red': ['cherry', 'pomegranate']}

Lista – dodatnie ujemne z list:

