Week 1

Sunday, June 10, 2018 10:43 AM

>>>	The prompt	We type **Python instructions**

- **Python instruction examples:** 2 + 3 is also a mathematical **expression**
- **<variable> = <expression>** Python instruction
- When we type 2 + 3 (**addition operation)** and we hit enter, Python will **evaluate the instruction**

Operator	**Symbol**	**Description**
Addition	**+**	
Subtraction	-	
Multiplication	**•**	
Exponentiation	******	We read this (2 ** 5) expression as **2 to the power of 5**
Division	**/**	Every division using **/** results in a **float**
Integer division	**//**	Example: 5 / 3 = **[1]** 2/3 = **[1] -** Integer division returns a whole number
Remainder (Modulo(Mod))	**%**	Example: 5 / 3 = 1 **[2]**/3 = **[2] -** Remainder (Mod) returns the remaining value in a division

Types	**Description**
int	Integer
float	Floating point number - floating point numbers are approximations of real numbers

- Python has a limited amount of memory to work with. When we type 2/3 **Python evaluates** to a finite position: for example the result for this evaluation is 0.6666666666666666 to a 16th position (decimal point)

>>> 3 + 4 - 5	Is **an expression** • Operations are performed from left to right: First 3 is added to 4 and then 5 is subtracted from the result
>>> 4 + 5 * 3	Multiplication should be applied first
>>> -10	The - sign we use for subtraction can be used with a single number in order to **meaning Negation**

Order of precedence (operation)

Operator	**Symbol**	**Precedence**
Exponentiation	******	Highest precedence
Negation	-	
Multiplication	**•**	
Division	**/**	
Int Division	**//**	
Mod (Remainder)	**%**	
Addition and Subtraction	**+ and /**	Lowest precedence

- Just like in math we can override operator precedence using parentheses

Operation between parentheses are evaluated first	
5 + 3 * 2	11
(5 + 3) * 2	16

Syntax: The rules that describe valid combinations of **Python symbols**
Syntax rules: The rules that specify which combination of symbols are legal

- When we asked python to evaluate the expression 2 + 3 it yielded a result (5)
  - 2 + 3 follows the **syntax** of the Python language: So 2 + 3 is a valid **Python expression**

Syntax Error

>>> 3 +	**Is not a valid syntax:**

- When we ask Python to evaluate an **invalid Python expression** such as (**3 +)** we get an error:

>>> 3 + **Syntax Error: invalid syntax: <string> line, pos**	Python doesn't understand what to do with that combination of symbols
>>> 4 + 3) **Syntax Error: invalid syntax:**	
>>> (4 +	If we hit enter nothing happens: Python allows instructions to **span multiple lines**. Therefore when we type (3 + and hit enter, Python spans new lines waiting for the closing parenthesis, then the expression is evaluated

Semantic Errors

Semantic	Relating to meaning in language or logic
Semantic Errors	Semantic errors occur when the meaning of a particular expression is invalid
2 + 3	This **syntax** example is valid. That is a valid **combination of symbols** and the **semantics** of this expression is that two is added to three
>>> 4 / 0 **Builtins.ZeroDivision.Error: division error**	Is valid syntactically, in that we are able to use this combination of symbols. However, the meaning of this expression is invalid:

When a spreadsheet program calculates the average of a group of numbers, the program first adds all the numbers then counts how many numbers are there, and then does the appropriate division:	All these values (resulted from the steps) are stored in computer memory

Memory Addresses: Think of the **computer memory** as a very long **list of storage locations**

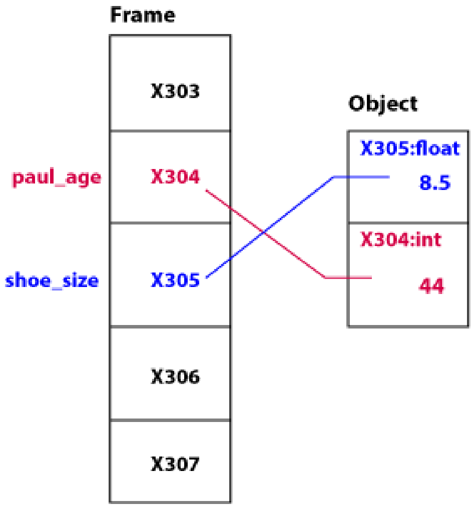
- Each storage location has a unique number called **memory address**
- We usually write memory addresses with an **X** as a prefix so to look different than other numbers
- **X123** is a memory address number
- **Values** are store in computer memory
  - Programs have ways to keep track of these values
    - **Using Variables** programs keep track of these values

VARIABLES

Variable: A named location in computer memory

- Variables are used to remember results generated from the evaluation of specific expressions in order to later use those values.

- Programs use **variables** to keep track of **values stored** in **memory addresses**
- Python keeps track of a variable in a separate area of memory from the values



- We can have a **variable** shoe_size that **stores memory address** X34
- We store memory addresses in variables

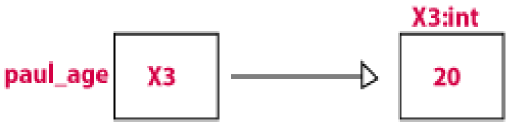
Terminology	• A value has a memory address • A variable contains a memory address • A variable refers to a value • A variable points to a value	
Examples of this terminology:	• Value 8.5 has memory address X34 • Variable shoe_size contains memory address x34 • Variable shoe_size refers to value 8.5 • Variable shoe_size points to value 8.5	

Assignment statement:

>>> base = 20	What this does is it evaluates the 20 (right hand side) and it associates that value with the variable base
Variable	Base it is called a variable because its value can vary
=	Equal symbol has different meaning in programming than in mathematics

- A notation for describing Variables and their Values
- Values in Python live in particular memory addresses

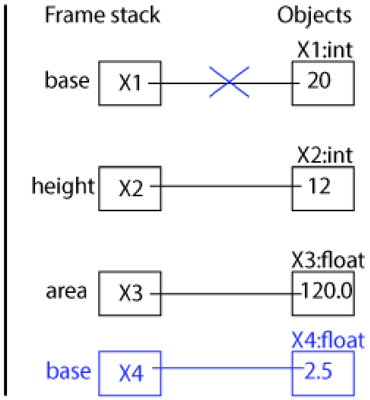
The Assignment statement takes the memory address X3 and puts it in the box associated with base -> and in some sense that means that it points to memory address X3 where the value 20 is:



>>> base = 20 : Variable -> base**[X3]** -> points to memory address **X3[20]** where the value 20 is
>>> height = 12: Variable -> height**[X4]** -> points to memory address **X4[12]** where the value 12 is

- We can use these variables in an expression such as:
  - Base * height / 2 = 120.0
  - 20   *   12 / 2 = 120.0
- We can assign the value of an expression to a variable, for example
- **>>> area = base * height / 2**
- **>>> area**   **area[X9]**   **------------->**   **X9:float[120.0]**
- **120.0**

```
>>> base = 20
>>> height = 12
>>> area = base * height / 2

>>> base = 2.5
```



- Every **Assignment Statement** has the form: **variable = expression**

Assignment Statement:
<variable> = <expression>

Rules for executing an assignment statement:

IMPORTANT:
Step 1: Evaluate the expression on the right of the = sign to produce a value. This value has a memory address
Step 2: Store that memory address of the value in the variable on the left of the = sign.

Rules for legal python names

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

If you start a variable with a number:

```
>>> 4_score = 4 * 20
SyntaxError: invalid syntax
```

```
>>> hours@noon = 12
SyntaxError: invalid syntax
```

Python is case sensitive

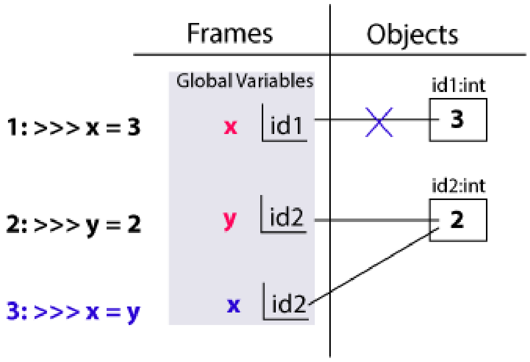
```
Hours_to_go = 12
hours_to_go = 13
```

Both are different variables

Python naming conventions:

- Use **pothole_case** so other Python programmers have easier time reading my code

Visualizing Assignment Statement



- Value inside an object in computer memory
- Variable x is going to contain the memory address of that value
- The **first assignment** X referred to value 3 and **the last assignment x** refers to value two

<variable_name> = <expression>

- Evaluate the right hand **<expression>** which gives back a **memory address** and stores it in **<variable_name>**
- The assignment statement creates a new variable OR changes the value of a variable

Built-in Functions

- A lot of operations but not enough symbols
  - Python has built-in functions that allow us to perform these operations

>>> <function>(x_(argument)_**, y**_(argument)_**)**

Argument: A value given to a function:
Pass arguments: To provide argument(s) to a function
Call: Ask Python to evaluate a function

Form of a Function Call

function_name(arguments)

1. The name of the function
2. Open parentheses
3. A comma separated list of expressions known as arguments
4. Closing parentheses

- When a function is called Python first evaluates the arguments then calls the function.

Rules for executing a function call

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

Return: pass back a value

We can find out which built-in functions are available by using another built-in function **dir.**

- **dir(__builtins__) - Call dir and ask for a listing of built-in functions**

***** The size of an empty string variable is 25 bites, then every position increases by one bite**

Another SyntaxError:
>>> m = "some string
SyntaxError: EOL (end of line) while scanning string literal

Week 1 - lecture(shell)

Sunday, June 10, 2018 8:16 PM

Variables

Assignment Statement

<variable_name>> = <expression>

How it is executed?

Variable	**=**	**Expression**
LHS (left hand side)		**RHS (right hand side)**

1. Evaluate expression on the RHS to produce a **value;** this value **has a memory address**
2. Store that memory address in the **variable** on the LHS
   a. If the variable exists update its value, otherwise create a new variable

Let's check a statement:

X = 7 --------------> X[id1] -----> id1:int[7]

We say:

- "X gets 7"
- "X refers to the value 7"
- "X contains memory address id1"
- "Memory address is stored in variable X"

Variable Names

- Must start with a **letter or underscore**
- Can include letters, digits, and underscores, but nothing else

Case matters

```
 >>> age = 11
>>> aGe  # Error! This is not defined
```

Conventions for the format of Names

- Python conventions: **pothole_case**
- **CamelCase** is sometimes seen, but not for functions and variable names.
- Python cares only about the format, not the content of the names (Python does not understand language (English).
- For example if you are **adding something**, *total* is better than *x*

Python Types

Int	**Integers**
Float	**Floating point numbers: Floating point numbers are an approximation of real numbers**

Check type:

```
>>> type(3)
>>> <class "int">
>>> type(3.0)
>>> <class "float">
```

Operator	Symbol	Description
Integer Division	//	Returns a whole number
Remainder pronounced Mod or Modulo	%	Returns the remainder
Exponentiation	**	Means to "To the power of"

Order of precedence

Operator	Symbol	Precedence
Parentheses	(expression)	Highest
Exponentiation	X ** Y	.
Negation	- X	..
Multiplication *(left to right)*	X * Y	...
Division *(left to right)*	X / Y	....
Integer Division *(left to right)*	X // Y	.....
Remainder (Mod) *(left to right)*	X % Y	......
Addition and Subtraction	X + Y - Z	Lowest

Errors (explains what)

```
>>> 4 +
SyntaxError: invalid syntax
>>> 4 + 5)
SyntaxError: invalid syntax

>>> 4 / 0
Builtins.ZeroDivisionError: division by zero

>>> min(1, 2, 3)
1
>>> max(5, 2, 1)
5

 dir(__builtins__)

>>> abs(-3.4)    -----> abs() is a built-in function
3.4
>>> help(abs)
        Help on built-in function abs in module builtins
```

*** Iterable are "things" that can be iterated over ***

```
round(float, int)
>>> round(1.5, 1.5)
```

Builtins.typeError: float object cannot be interpreted as an integer

Week 2

Tuesday, June 12, 2018 10:21 PM

Defining own Function

Example from math:

f(x) = x^2

```
---------------------
def f(x):
  return x ** 2
--------------------------
```

def:	A keyword indicating a function definition
f:	The name of the function
x: The parameter:	A variable that appears between parentheses of a function definition. Parameters get their values from **expressions** *(2 + 3)* in a **function call** *(function_name(argument(expression))*
The colon (x)**:**	Indicates to Python that we are about to type what happens when the function is called
return:	• A keyword indicating the result of a function • Indicates that we are passing back a value

Return Statement

Form of return statement	**return <expression>**

Rules for executing a return statement

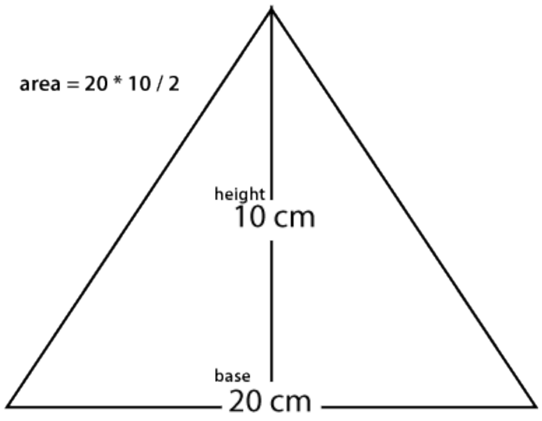
1. Evaluate the expression, which produces a value (memory address)
2. Pass back that value (memory address) to the caller / Produce that value as the result of the function call

Function Definition

Form:	**def function_name(parameter):** **body** *(notice that the body is indented (4 spaces))*
Step by	1. The word **def** followed by the name of the function **function_name** **2.** Zero or more **parameters** *(parameter, parameter)* separated with comas. **3.** The **body** of the function, which is one or more statements often ending with a **return** statement

- Lets call the function **f** passing the argument *(f(**argument**))* **3:**
  - **f(3) <=> x = 3 (Assignment statement)**
    - When the function is called, the **parameter x** is assigned the memory address of the value 3 | We can think of it as an assignment statement -> **<variable> = <value>**
- Function calls are expressions, so we can use a variable to store the result

Let's create a function to calculate the area of a triangle



Write the function	**def area(base, height):**
	return base * height / 2

Execute (area) by passing two arguments	**>>> area(20, 10)** The expression is evaluated as base times height and the result is returned

Function Call

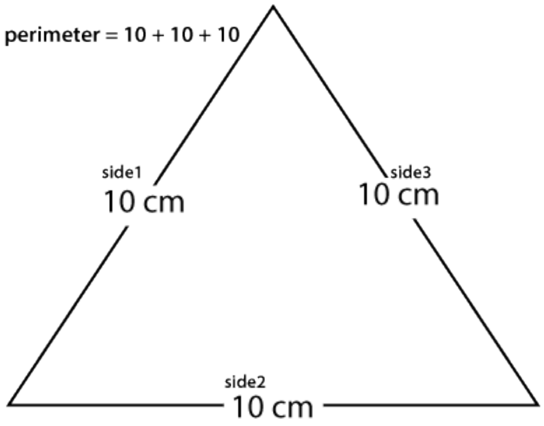
Form	**Function_name(arguments)**

Rules for executing a function call

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

- **Most Python programs are saved in files**
- **We create Python files and save our function definitions in them**
- *****These files are also called modules*****

Lets create a function to calculate the perimeter of a triangle



Write the function	**def perimeter(side1, side2, side3):**
	return side1 + side2 + side3

Execute (area) by passing two arguments	**>>> perimeter(10, 10, 10)** The expression is evaluated as side1 + side2 + side3

*****Before executing a function in the shell we need to run the module (the Python file) containing the function we created and stored in the Python file**

Type str: Strings in Python

Python's String representation

str:	**Python's string type**	• Starts and ends with single quote (') or double quotes ("). • Strings are values, therefore can be used in assignment statements	
String literal	**Sequence of characters**	• We **cannot** use single quote(s) inside single quoted strings or double quote(s) inside double quoted strings • We can do so by using **escape character.** For example "She said **\\"**Hi**\\"** to you"	

Escape Character: \

- The character following the escape character is treated differently from normal

Escape Sequence: \\"

- The escape character with the character that follows it

Concatenation

Expression	**Description**
str1 + str2	Concatenate str1 and str2

When concatenating strings we have to put in our own spaces	Example: if we type **'name' + 'surname'** we get **namesurname,** without space in between
We can build up a repeated string using the multiplication operator:	Example: **string1 * int** concatenates **int** copies of **string1** Example: **int * string1** the equivalent of the statement above

- The * and + operators obey the standard precedence rules when used with strings
- *** ALL other mathematical operators and operands result in **TypeError** when used with strings

>>> "This is a string" + **5.5**	**Builtins.TypeError: must be string not float**

>>> "ha" * "5"	**Builtins.TypeError: can't multiply sequence by non-int of type str**
>>> "a" - "b"	**Builtins.TypeError: unsupported operant type(s) for -: 'str' and 'str'**
>>> "a" / "b"	**Builtins.TypeError: unsupported operand type(s) for /: 'str' and 'str'**
>>> "a" ** "b"	**Builtins.TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'str'**

*****NOTE***** Every error above is a **TypeError**

Input/Output and str formatting

Python's built-in function	**print()**	function(argument) = print("Hello")	Prints a sequence of arguments for the user to read. Multiple arguments are separated by spaces

Command in the shell	**Output**	**Description**
>>> print("Hello")	Hello	Notice how Hello is printed without the quotes "". The quotes are only for Python's *internal string representation* and they're not seen by the user
>>> print("Hello", "There")	Hello There	Notice the space between: Multiple arguments *(separated by comas)* are separated by spaces

1. Create file **calc.py**
2. **Define functions**

Function 1	def square_return(num:int) -> int:
	    return num ** 2

Function 2	def square_print(num: int) -> int: print("The square of num is", num ** 2)

3. Run module/file calc.py

Command in the shell	Output	Describtion
>>> answer_return = square_return(3)		
>>> answer_return	**9**	answer_return has a numeric value therefore we can treat it as such. For example: answer_return * 2 = 18 But, the answer_print below is of NoneType, therefore when we execute answer_print * 5 we get the following error: **Builtins.TypeError: unsupported operand type(s) for *: 'NoneType' and 'int'**
>>> answer_print = square_print(3)	**The square of num is 9**	**The value is printed while assignment statement**
>>> answer_print	**None**	• During execution of a function call, if the end of the function body is reached without executing a return statement, **that function call produces a value None**

*****Value None has NoneType*****

<u>Built-in:</u>
<u>Input:</u> Get information (string) from the user

input("argument")	- The argument to input is called a prompt - The function pauses until the user types a newline - The return value is the string the user typed

Command in shell	Output	Describtion
>>> Input("What is your name ?"	**What is your name?**	If the user replies with **Name,** the result in the shell will be **'Name'**, with quotes, indicating that the returned/inputted by the user value is a string.
>>> name = input("What is your name? ")	**What is your name?**	If the user replies with **Name,** the result will be stored in the variable name. Look below when we ask Python to return the value of the variable name
>>> name	**'Mondi'**	The variable **name** has been assigned the memory address that contains the string 'Mondi'

*****All values returned by input are Type String*****

Command in shell	Output	Describtion
>>> num_coffe = input("How many cups of coffee have you had today? ")	**How many cups of coffee have you had today**	Watch what happens next if the user enters the value **2**
>>> num_coffee	**'2'**	Notice the quotes indicate that the value returned is of type string

<u>String format - Triple quotes string</u>
***** Triple quote strings can span multiple lines*****

Command in shell	Output	Description
>>> """Hello"""	**'Hello'**	
>>> print("""How …are …you?""")	**How are you?**	Notice how the text is printed in multiple lines
>>> s = """How …are …you?"""		Nothing returns since we are assigning a value
>>> s	**'How\nare\nyou?\**	Notice the **\n**
>>> print('3\t4\t5)	**3 4 5**	Notice the space (tabbed) in between the digits
>>> print('\'')	**'**	Notice how only one quote is printed
>>> print("\")	**SyntaxError**	**SyntaxError: EOL (end of line) while scanning string literal: <string> line, position**

<u>Escape sequences:</u>

\n	Newline	ASCII linefeed - LF
\t	Tab	ASCII horizontal tab - TAB
\	Backslash	The **escape character**

\n	Backslash plus n	The **escape sequence**

Docstring and Function Help

Ex: Built-in function 'help'

>>> help(abs)	When you call function ***help(abs)*** on the abs function you get a description of what the function does

- You can provide same kind of information for your own functions by writing documentation in a **particular format called a *docstring***

Docstring example in Function when writing a function:

```
def mul(num: float) -> int:
    """ return the sum of  num multiplied by 3

    >>> mul(3)
    9
    """
    Return 3 * num
```

- We use a triple quoted string because our description spans multiple lines.
- We have to always provide docstrings for our functions both to describe what they and also to hook into python's help system.

Function Design Recipe

1. **Header**
   a. Function Name
   b. Parameters
   c. Type contract:
      i. Parameter types:
      ii. Return type
2. **Describtion**
   a. Docstring: What it does
3. **Examples**:
   a. How to use the function
4. **Body**

- **Function design recipe helps us develop these four parts plus one more when we test that our function works**

Designing the Function

The Problem: USA uses Fahrenheit to measure temperature, while Canada uses Celsius. We will write a function that converts from Fahrenheit to celsius

The Recipe:

1. **Examples**
   a. Think what should your function do?
   b. Type two example calls: What should they return?
   c. Pick a name (often a verb or a verb phrase because they do things)
      i. So if you are stuck, answer the question "What does the function do?"

```
>>> convert_to_celsius(32)
0.0
>>> convert_to_celsius(212)
100.0
"""
```

*****These examples are going to become part of our docstring and we mark that by putting triple quotes around them*****

2. **Header**
   a. Write **def** and name of the function

def convert_to_celsius():	Parameters will go between the parenthesis and always end with a colon We know that our parameter is the number of degrees Fahrenheit, so we chose **fahrenheit** as parameter name: ***Pick meaningful parameter names***

def convert_to_celsius(**fahrenheit):**	Now that we typed the parameter we have to fill in the type contract: What are the parameter types? **Float** What type of value is returned? **Float**
def convert_to_celsius (fahrenheit: **float) -> float:**	

3. **Describtion**

"""Return the number of Celsius degrees equivalent to Fahrenheit degrees

- **Mention every parameter in your description**
- **Describe the return value**

4. **Body**

return (fahrenhreit - 32) * 5 / 9

-------------------All Together---

Header	def convert_to_celsius(fahrenheit: float) -> float:
Description	**"""Return the number of Celsius degrees equivalent to Fahrenheit degrees**
Examples	**>>> convert_to_celsius(32)** **0.0** **>>> convert_to_celsius(212)** **100.0** **"""**
Body	**return (fahrenheit - 32) * 5 / 9**

Command in shell	Output	Description
>>> convert_to_celsius(212)	100	Test results as expected

Recipe for Designing Functions - Step by Step

1. **Examples**
2. **Header**
3. **Describtion**
4. **Body**
5. **Test**

Save File in temperature.py

Function Reuse

- Once defined, a function can be used as many times as we want. We can call it from within other function definition and from other function calls

Triangle.py

-------------------------area of a triangle--

```
def area(base: float, height: float) -> float:
    """Return the area of a triangle with dimensions base and height.

    >>> area(10, 20)
    100.0
    >>> area(6.4, 9.5)
    30.4
    """

    return base * height / 2
```

-----------------------perimeter of a triangle-------------------------------------

```
 def perimeter(side1: float, side2: float, side3: float) -> float:
    """Calculate the perimeter of a triangle with sides of length side1, side2, and side3.
    >>> perimeter(15, 15, 15)
    45.0
    >>> perimeter(12, 13, 13)
    48.0
    """

    return side1 + side2 + side3


---------------------semi-perimeter of a triangle---------------------------------
 def semiperimeter(side1: float, side2: float, side3: float) -> float:
    """Return the semiperimeter of a triangle with sides of length side1, side2, and side3.

    >>> semiperimeter(15, 15, 15)
    22.5
    >>> semiperimeter(12, 13, 13)
    24.0
    """

    return perimeter(side1, side2, side3) /2
----------------------------------------end file triangle.py---------------------------
```

- **By re-using the perimeter function we reduce the risk of error when calculating the semiperimeter since that function (the perimeter) has already been tested.**

Solving another Problem

2 Slices of Pizza left. Find which slice is bigger - Since slices are triangles we can measure base and height of the slices.

We reuse the area function:

max(area(5, 75), are(6, 5.8)

- **Not only can we pass function calls to function definitions, but we can also pass function calls to other function calls, because function calls themselves are expressions.**

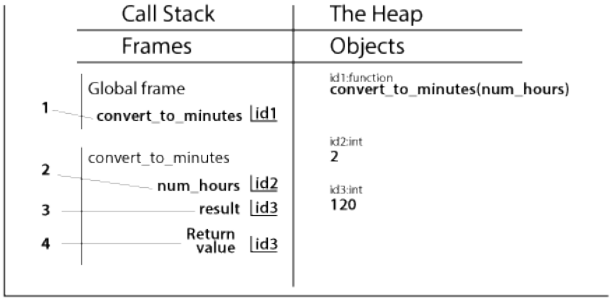
Function Calls

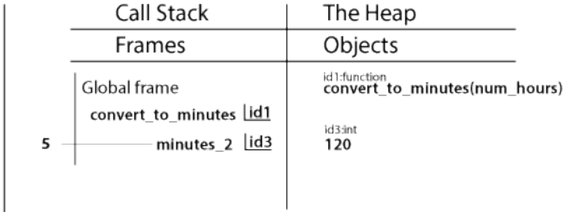
How function calls are managed in computer memory

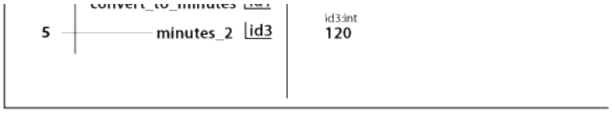
```
------------------------------------------------------------------------------------------
def convert_to_minutes(num_hours: int) -> int:
    """Return the number of minutes there are in num_hours hours

    >>> convert_to_minutes(2)
    120
    """

    result = num_hours * 60
    return result
---------------------------------------------------------------
```

- **Let's visualize the executing of the above function**

```
1 -> def convert_to_minutes(num_hours: int) -> int:
    3 -> result = num_hours * 60
    4 -> return result

    2 -> minutes_2 = convert to minutes(2)
```







1. **Variable convert_to_minutes is created.**
   It contains the memory address of a function object. That function object contains all the information about the function, including any code that needs to be executed, the parameter(s) and the docstring.
2. **Variable num_hours is created pointing to value 2**
   An Assignment statement: Notice that the variable minutes_2 has not been created yet. That will happen after we finish executing the assignment statement: So first we have to evaluate the expression on the RHS which is a function call -> function(argument). Once that function call has returned it will give us back a value and then minutes_2 will be created and it will store the memory address of that returned value. In order to execute this function call, the argument is evaluated. (2) that will create a box for value 2. In addition a new region of memory will be created to keep track of what happens while convert_to_minutes is executed: Step two executed
3. **Assignment statement**
   Evaluate RHS - num_hours is 2. New variable result is created inside the Function Frame for the function convert_to_minutes pointing to the value 120
4. **Return value is created**
   Return value created inside function frame pointing to the same id value that variable result points to, since we are returning the variable result
5. **Minutes_2 is created**
   After RHS is evaluated (convert_to_minutes(2)) the value is passed back to variable minutes_2, minutes_2 is created and added to the call stack Main Frame (global frame). Stack frame for the call convert_to_minutes is gone from com. Memory will all temporary/local variables. That happens when a function exits

Left hand side area**: Call Stacks / Frames Stack**	**Stack frame: A region of computer memory for keeping track of information about a function being executed** • The area to the left is called the Call stack (like an upside-down stack of plates. • When a function is called, a **stack frame** is created, and when that function returns its frame is removed from the stack. • Each Frame contains variables that are specific to that section of the program. • **The main frame contains variables that are created outside of the main function** • **Function frames contain parameters and Local Variables** ○ **Local Variables are variable created inside function body**
Right hand side are: **The heap / Objects**	The side to the RHS of computer memory is called The Heap. It contains all objects and contains all values that are created during execution of a program

Constants

```
def is_pass(grade: float) -> bool:
    """Return True if and only if the grade is a passing grade.

    >>> is_pass(80)
    True
    >>> is_pass(30)
    False
    """
    Return grade >= 50
```

- We say 50 is a magic number. Its this value that occurs in the function for which we haven't explained the meaning. Sometime we see the same magic numbers being used multiple times in our program. We are going to **replace that number with a CONSTANT**

PASS_BOUNDARY = 50

- We use uppercase letters to indicate that is a constant. What makes it a constant is that its value should not be changed in the program. That is what the uppercase means

Now we use: return grade >= PASS_BOUNDARY

Week 2 - lecture (shell)

Wednesday, June 13, 2018 11:02 AM

>>> 333_variable = 5	SyntaxError: Invalid token
>>> hello-bye = 3	SyntaxError: can't assign to operator:
>>> max = 10 >>> max(10, 20)	Builtins.TypeError: 'int' object is not callable

```
def double(num: float) -> float:
        """Return twice the value of num

        >>> double(2)
        4
        >>> double(4)
        8

        num * 2
```

```
>>> double(7)
>>> result = double(7)
>>> print(result)
None
>>> type(result)
<Class 'TypeNone'>
```

If we change num * 2 to print(num * 2)

```
>>> double(7)
14
>>> Result = double(7)
14
>>> type(result)
<Class 'TypeNone'>
```

Week 3

Saturday, June 16, 2018 12:45 PM

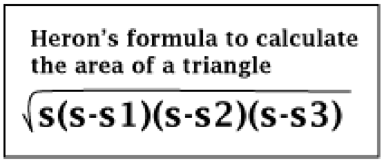
Import: Using Non-Builtin Functions

- Python has hundreds of functions and most are not immediately available as built-ins. Instead the functions are saved into different modules and you need to **import them** when you'd like to use them
- Similarly as we saved our **own functions** into a different module (i.e. triangle.py)

MODULE: A file containing function definitions and other statements

A python **(.py**) defines a module; **triangle.py** defines module triangle

- Task: *Write a function to calculate the area of a triangle given the length of three sides using Heron's formula.*



Heron's formula to calculate the area of a triangle

$\sqrt{s(s-s1)(s-s2)(s-s3)}$

1. We need the semiperimeter; we have already defined a function in triangle.py for calculating the semiperimeter.
2. We also need a function to calculate the **square root**
   a. Such function exists but isn't a built-in function. It is defined in another file **(module)** named **math**.py
   b. *tirangle.py is a module just like math.py*

To see a listing of the math module let's **import the module to gain access to it and then we'll call the built-in function dir(math).**

- We see that a function named sqrt is in the module.

>>> help(**math.**sqrt)	To ask help on the sqrt function we need to specify that is in the math module

Now we are ready to write the function:

```
def area_heron(side1: float, side2: float, side3: float) -> float:
    """Calculate the area of a triangle with sides of length side1, side2, and side3.
    """
    semi = semiperimeter(side1, side2, side3)
```

Now that we have imported the math module (import.math)

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

<u>Boolean values</u>

<u>Python Type Bool</u>

Comparison Operator	**Symbol**
Less than:	**<**
Greater than:	**>**
Equal to:	**==**
Less than or equal to:	**<=**
Not equal to:	**!=**

Shell	**Result**	**Description**

>>> 3 < 4	**True**	When this expression is evaluated we get a True or False value, the type of value we get is type bool
>>> 7 == 7	**True**	Equality operation: Two equal signs signify equality since the one (=) equal sign is used for the assignment operation
>>> 7 == 7.0	**True**	

>>> x = 7
>>> y = 8
>>> x == 8
	False	

>>> 3 != 4	**True**	Inequality operator. We can check whether 3 is not equal to 4

The comparison operator take two values and return a Boolean value, either True or False

Python has three **logical operators.** Applied to boolean values and yield boolean results.
Logical Operator: Operands are Boolean expressions

Logical operator	**Symbol**	**Order of precedence**
Not:	**not**	Highest
And:	**and**	.
Or:	**or**	lowest

Shell command	**Result**	**Description**
>>> grade = 80 >>> grade >= 550	**True**	
>>> **not** (grade >= 50)	**False**	First, the expression inside parentheses is evaluated and that gives the value True and then **not** is applied to true. Something that is **not True** is **False.** So we get back False
not not (True) == True	**True**	

<u>Truth Tables Logical Operands for reference:</u>

<table>
<tr><td><b>If A then B |Implication| A &gt; B</b></td><td></td><td></td><td></td><td><b>Not A |Negation| -A</b></td><td></td></tr>
<tr><td>TT -&gt; T</td><td></td><td></td><td></td><td>If A is true -A is false</td><td></td></tr>
<tr><td>TF -&gt; F</td><td></td><td></td><td></td><td>If A is false -A is true</td><td></td></tr>
<tr><td>FT -&gt; T</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>FF -&gt; T</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td><b>A or B |Disjunction| A v B</b></td><td></td><td></td><td></td><td><b>A and B |Conjunction| A &amp; B</b></td><td></td></tr>
<tr><td>TT -&gt; T</td><td></td><td></td><td></td><td>TT -&gt; T</td><td></td></tr>
<tr><td>TF -&gt; T</td><td></td><td></td><td></td><td>TF -&gt; F</td><td></td></tr>
<tr><td>FT -&gt; T</td><td></td><td></td><td></td><td>FT -&gt; F</td><td></td></tr>
<tr><td>FF -&gt; F</td><td></td><td></td><td></td><td>FF -&gt; F</td><td></td></tr>
</table>

Converting between int, str, and float

Str(): return the information <u>wrapped up</u> in the argument as a **string.**
Int(): return the information in the argument wrapped up as an integer.
Float(): return the information in the argument wrapped up as a float.

Shell command	**Result**	**Description**
>>> str(3)	**'3'**	The result is number three as a string
>>> three = str(3) >>> three	**'3'**	
>>> three * 3	**'333'**	Since it's a string we can multiply it.
>>> m = 'Mondi' >>> int(m)	**Error**	Builtins.ValueError: invalid literal for int() with base10.

<u>Use of conversion</u>

```
>>> input("Enter the number of shoes: ")
Enter the number of shoes: 10
>>> num_shoes_left = 9
>>> num_shoes_wanted = int(input("Enter the number of shoes: ")
>>> num_shoes_left >= num_shoes_wanted
False
```

IF Statement

<u>**Simple form of if , elif, and else statements**</u>	<u>**Description**</u>
If expression: **statement**	
elif expression: **statements**	
else: **statements**	Else says: If none of the preceding conditions are True, do this:

- The programs we've written so far execute the same sequence of instructions each time they run.
- We are going to use boolean expressions to control which instructions get executed

<u>**The problem**</u>**:** *A flight was scheduled to arrive at a particular time and it is now estimated to arrive at another time.*

- *Write a function that returns the flight status:* ***on time, early, or delayed.***

Times will re represented as float. Example: **Time 3:00 will be 3.0 and time 14:30 will be 14.5**

Precondition: The conditions under which a function is intended to work:
Precondition: **0.00 (inclusive) - 24.0 (exclusive)**

```
 def report_status(sched_time: float, estimated_time: float) -> str:
    """Return the status of the flight (early, on time, or delayed) for flight scheduled to arrive at sched_time time but it is n ow estimated to
    arrive at estimated_time time

    Precondition: 0.0 <= sched_time < 24 and 0.0 <= estimated_time < 24
    >>> report_status(14.0, 14.0)
    'on time'
    >>> report_status(14.0, 13.0)
    'early'
    >>> report_status(14.0, 15.0)
    'delayed'
    """
    if sched_time == estimated_time:
        return 'on time'
    elIf sched_time > estimated_time:
        return 'early'
    else:
        return 'delayed'
```

*If a function ends **without a return** statement being executed, the function returns **None***

<u>**General features of If statements:**</u>

- We can have 0 or more elif clauses associated with an if
- We can have 0 or 1 else clauses and it has to be the last clause for the if statement

When we have if statements with these various expressions we evaluate each one in order: The first expression that evaluates to true, its body it's executed and the function exited without checking any further conditions.

<u>**Style issue when you combine boolean functions and if statements:**</u>

```
 def is_even(num: int) -> bool:
    """Return whether num is an even number.
    >>> is_even(8)
    True
    >>> is_even(7)
    False
    """

    If num % 2 == 0:
        return True
    else:
        return False
```

The Problem here is that the function body is 3 lines too long. Since num % 2 == 0 is already a boolean expression, the if and else statements are redundant.
We don't say If something is true than return true, otherwise return false.

*We wanna **return** a boolean: since **num % 2 == 0** already produces boolean, so we **return num % 2 == 0.***

Structuring If statements

Two problems while deciding how to construct our code.

First problem: if grade is passing grade

```
if grade1 >= 50:
    print("You passed with: ", grade1)
elif grade2 >= 50:
    print("You passed with: ", grade2)
```

*One boolean expression associated with the **if** and another with the **elif** statement. These expressions are evaluated from top to bottom and, once one of the expressions is True its body gets executed and the if statement exits at that point without evaluating the subsequent expressions. Because of this, at most one of the two statements will get executed.*

```
if grade1 >= 50:
    print("You passed with: ", grade1)
if grade2 >= 50:
    print("You passed with: ", grade2)
```

*Since the **first if** statement evaluated to True, we jumped to the next if statement.*
The conditions are the same but because the second two statements are if and if as opposed to if and elif statements it behaves differently.
We first evaluate the expression associated with the if and its true so we enter into its body and print the message, then we exit the statement and we move to the next line of the code where there is another if statement which is also true and its body is executed as well.

So if and elif are not equivalent to if and if statements.

Second problem: Should we bring an umbrella or not?

```
precipitation = True
temperature = +8

if precipitation:
    If temperature > 0:
        print("Bring your umbrella")
    Else:
        print("Wear a coat and bring winter boots")
```

*The first if is true so we enter the body where we have a **nested if statement** which also evaluates to true*
We want to write a version without nesting: We can use boolean operator and to structure it as a single if statement.

```
if precipitation and temperature > 0:
    print("Bring umbrella")
elif precipitation:
    print("Wear a coat and bring winter boots")
```

Both functions are equivalent.

More str Operators

Shell command	**Result**	**Description**
>>> solution = 'cat'		Assignment statement
>>> solution == 'cat'	**True**	We ask if variable solution equals to the string 'cat'
>>> 'abc' < 'acc'	**True**	Compare two strings for their **dictionary order.** Here Python compares letter by letter starting at the beginning
>>> 'A' >= 'A'	**True**	
>>> 'A' < 'a'	**True**	Capitalization matters: Lowercase in comparison results greater than uppercase. So, capital letters are less than lowercase letters
>>> 's' == 3	**False**	We can compare for equality different types, but not for ordering, as follows:
>>> 's' >= 3	**TypeError**	builtins.TypeError: '>=' not supported between instances of 'str' and 'int'

Operator	**Symbol**	**Description (or example)**
Equality:	**==**	Is **len('Mondi') ==**(equal to) **5 -> True**
Inequality	**!=**	Is **len('Mondi') ==**(NOT equal to) **5 -> False**
Less than	**<**	**3 < 2 -> False**
Greater than	**>**	**3 > 2 -> True**
Less than or equal to:	**<=**	**2 <= 2 -> True**
Greater than or equal to:	**>=**	**3 >= 3 -> True**
Contains:	**In**	**'a' in 'a**eiou**' -> True**

Shell command	**Result**	**Description**
>>> 'C' **in** 'abc'	**False**	We check if the capital letter C is contained in the string 'abc'
>>> 'zoo' **in** 'ooz'	**False**	Order of characters matters.
>>> 'a' **in** 'abc'	**True**	The letter lowercase **a** is in the string **'abc'**
>>> '' **in** 'anystring'	**True**	Empty string **"" is** always present within a string
>>> len(str('ha' * 2))	**8**	Len(str) return the number of characters in ('string')

- Values of different types can be compared for equality
- Values of different types usually cannot be compared for ordering. Although, **int and float can.**

str: Indexing and Slicing

- When working with strings we often need to **extract substrings**
- Two techniques, **Indexing and Slicing**

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
L	e	a	r	n		t	o		P	r	o	g	r	a	m
- 16	**- 15**	**- 14**	**- 13**	**- 12**	**- 11**	**- 10**	**- 9**	**- 8**	**- 7**	**- 6**	**- 5**	**- 4**	**- 3**	**- 2**	**- 1**

*Each string has an **index:** Position within a string*

- *16 is the len(string) not index positions*

Shell command	**Result**	**Description**
>>> s = 'Learn to Program'		Assignment statement
>>> s[0]	**'L'**	The string at index 0 indicated using bracket notation
>>> s[1]	**'e'**	

>>> s[2]	**'a'**	
>>> s[-1]	**'m'**	-1 is always the last index
>>> s[-2]	**'a'**	
>>> s[-3]	**'r'**	

- So, using index we can extract parts of the string one character at a time
- Notice that **0 is the first position** (we start counting from 0)
- We can also use negative **indices** to count from the end, or the RHS of the string
- So using **indexing** we can extract parts of the string **one character at a time.**
- But using **Slicing,** we can extract more than one character at a time

Slice: A substring of a string from a starting index up to but including an end index
Slicing: str[start_index : end_index]

Shell command	**Result**	**Description**
>>> s[0:5]	**'Learn'**	We are **slicing:** So we are extracting indices 0 to 5(not included)
>>> s[6:8]	**'to'**	
>>> s[9:len(s)]	**'Program'**	• Start from index 9 and end to the length(string) which is 16 - meaning 16 positions* • This is also equivalent to >>> s[9:16]
>>> s[9:]	**'Program'**	Another option is to omit the ending index which by default case is to go to the end of the string
>>> s[:5]	**'Learn'**	The starting index can also be omitted
>>> s[:]	**'Learn to Program'**	We can also omit both starting and ending indices and gives the entire string

- Negative indices can be used for indexing and slicing

Forms:
Indexing: String[index]
Slicing: String[start_index : end_index]

Equivalent slicing:

>>> s[1:8]	'earn to'
>>> s[1:-8]	'earn to'
>>> s[-15:-8]	'earn to'

- The slicing and indexing operation do not modify the string, they act on. The value of s is unchanged by the above operations .

Strings are immutable: They cannot change

>>> s[6] = 'd'	Builtins.TypeError: 'str' object **does not support** item assignment
>>> s[9:16] = 'run'	Builtins.TypeError: 'str' object does not support **item assignment**

But we can add (Concatenate)

>>> s[:5] + 'ed' + s[5:]	'Learn**ed** to Program'
>>> s = s[:5] + 'ed' + s[5:]	*Assignment statement (We* ***did not modify*** *the original value, instead we* ***created a new string*** *and have variable s refer to it*
>>> s	'Learn**ed** to Program

Week 3 - Lecture (shell)

Tuesday, June 19, 2018 7:37 PM

- Recall order of function evaluation:

f(g() + 3, h())

- How many parameters does f, g, and h have?
- How many parameters does f have?
  Answer: 2 -> g()+3 and h()
- How many parameters does g have?
  Answer: 0
- How many parameters does h have?
  Answer: 0

Let's talk a bit about strings:

>>> first_pet = 'cat'	
>>> second_pet = "dog"	I can use either single or double quotes
>>> third_pet = rat	builtins.**NameError**: name 'rat' is not defined
>>> my_pets = 'first_pet' + ' second_pet'	
>>> my_pets	'first_pet second_pet'
>>> my_pets = first_pet + second_pet	
>>> my_pets	'catdog'
>>> my_pets = first_pet + second_pet	
>>> my_pets	'catdog'
>>> my_pets = first_pet + ' ' + second_pet	
>>> my_pets	'cat dog'
>>> print(my_pets)	cat dog
>>> my_pets = print(first_pet, second_pet)	cat dog
>>> my_pets	**None**
>>> type(my_pets)	**'Class 'NoneType'**

Indexing a string: Finding the character at a position in a string

>>> my_pets[1]	**'a'**	That is index 1 at position 2
>>> my_pets[0]	**'c'**	That is index 0 at position 1
>>> my_pets[-1]	**'g'**	Negative indices counts backwards
>>> my_pets[6]	**'g'**	
>>> my_pets[:3]	**'cat'**	Index over ranges
>>> my_pets[0:3]	**'cat'**	Index over ranges

Slicing: Indexing over ranges

>>> my_pets[::2]	**'ctdg'**	This third number is called a **"stride"**
>>> my_pets[1::2]	**'a o'**	
>>> my_pets[::-1]	**'god tac'**	Negative stride
>>> 'at' **in** my_pets	**'True'**	Boolean operator -> x **in** y
>>> 'ta' **in** my_pets	**'False'**	
>>> phrase = 'Laughing Out Loud'		
>>> phrase[0]	**'L'**	
>>> phrase[0] + phrase[8]	**'LO'**	
>>> phrase[0] + phrase[8] + phrase[13]	**'LOL'**	Concatenation of slices
>>> phrase[::0]	**Error**	Builtins.ValueError: slice step cannot be zero

Boolean Card Game

Here's how the boolean card game is going to work:
I am going to type a boolean statement
True ==> Hold up GREEN card

False ==> Hold up RED card

Shell commands	Result	Description
>>> True	**True**	Because True is true
>>> False	**False**	Because False is false
>>> 4 > 3	**True**	Because 4 is greater than 3
>>> 4 > 4	**False**	Because 4 is not greater than 4
>>> 4 >= 4	**True**	Because 4 is greater or equal to 4
>>> 4 * 3 == 12	**True**	Because 4 times 3 equals 12
>>> 8 == 8.0	**True**	Because int and floats are equal if mathematical value is equal

Variable of type bool

>>> thursday = True		Assigning boolean value to variable
>>> rainy = False		
>>> type(rainy)	**<class 'bool'>**	*Boolean operators: **not, and, or**
>>> not rainy	**True**	Because **not False is True**
>>> not not rainy	**False**	Because not (not False = True) which means not True equals to false
>>> thursday and rainy	**False**	Because False and True == False
>>> True and True	**True**	Because True and True == True
>>> True and False	**False**	Because True and False == False
>>> thursday or rainy	**True**	Because One **or** The other == True
>>> monday = False		
>>> sunny = **not** rainy		
>>> sunny	**True**	Because **not rainy** when rainy == false, means **not False,** which means True
>>> not thursday or sunny	**True**	Because **not True** or True equals true. Considering order of precedence this expression can equals to **(not True) or True**
>>> not (thursday or sunny)	**False**	Since (**thursday or sunny == True) then not (thursday or sunny) == not True.** Which means False
>>> True or 1/0	**True**	Because the first operand evaluated to True, Python does not need to evaluate the second operand since the answer is already True **This behaviour of not evaluating the second operand is called short-circuiting (or lazy evaluation)**
>>> sunny and 1/0	**Error**	Builtins.ZeroDivisionError: division by zero Since the operand **and** requires both expressions to be true it evaluated
>>> monday = False		
>>> sunny = True		
>>> thursday = True		
>>> rainy = False		
>>> monday or rainy or sunny or thursday	**True**	Only one expression has to result in True when using boolean operator **or**
>>> thursday and sunny and Monday and rainy	**False**	Because all expressions have to be True when using boolean operator **and**
>>> time = 8		
>>> early = time < 12		
>>> early	**True**	Because early is True when time is less than 12, and time == 8, so it is True
>>> early or 3 > 2	**True**	

DON’T and DO - *Don't compare boolean values*

>>> 3 and 2 > 1	**True**	Sometimes people use this to **(WRONGLY)** mean: both 3 and 2 are greater than 1
>>> 0 and 2 > -1	**0**	Sometimes people use this to **(WRONGLY)** mean: both 0 and 2 are greater than -1
>>> (0 > -1) and (2 > -1)	**True**	This is the correct way to do it
>>> (0 and 2) > -1	**True**	Wrong: (0 and 2) what does this even mean?
>>> 'a' in 'cat'	**True**	
>>> True == True	**True**	
>>> True == False	**False**	
>>> 'a' in 'cat' == True	**False**	This is **WRONG**

>>> 10 > 3 > 1	**True**	
>>> 10 > 3 and 3 > 1	**True**	This is equivalent to the previous
>>> 'a' in 'cat' == True	**False**	Don't do this
>>> ('a' in 'cat') and ('cat' == True)	**False**	This is what the previous line is doing

Type('a')	<class 'str'>
Type(10.8)	<class 'float'>
Type(False)	<class 'bool'>