1 Expressions and Control Structures

Integers, Strings, and Operations

int (signed integers): They are often called just integers or ints, are positive or negative whole numbers with no decimal point.

long (long integers): Also called longs, they are integers of unlimited size, written like integers and followed by an uppercase or lowercase L.

float (floating point real values): Also called floats, they represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 102 = 250).

string (characters in quotes): Also called strings, they represent the literal characters that is encapsulated in double (or single) quotes. For example "abc123@%*" would be a string that literally takes in whatever the character is.

Know that you can set such primitive values to an arbitrary variable:

```
>>> a = 2
>>> a
2
>>> b = "abc123@%*"
>>> b
"abc123@%*"
```

Division

Let's compare the different division-related operators in Python.

True Division: /	**Floor Division: //**	**Modulo: %**

`>>> 1 / 5`
 `0.2`
	`>>> 1 // 5`
	 `0`
		`>>> 1 % 5`
		 `1`

`>>> 25 / 4`
 `6.25`
	`>>> 25 // 4`
	 `6`
		`>>> 25 % 4`
		 `1`

`>>> 4 / 2`
 `2.0`
	`>>> 4 // 2`
	 `2`
		`>>> 4 % 2`
		 `0`

`>>> 5 / 0`
 `ZeroDivisionError`
	`>>> 5 // 0`
	 `ZeroDivisionError`
		`>>> 5 % 0`
		 `ZeroDivisionError`

One useful technique involving the % operator is to check whether a number x is divisible by another number y:

```
x % y == 0
```

For example, in order to check if x is an even number:

```
x % 2 == 0
```

Boolean Operators

Python supports three boolean operators: and, or, and not:

```
>>> a = 4
>>> a < 2 and a > 0
False
>>> a < 2 or a > 0
True
>>> not (a > 0)
False
```

and evaluates to True only if both operands evaluate to True. If at least one operand is False, then and evaluates to False.

or evaluates to True if at least one operand evaluates to True. If both operands are False, then or evaluates to False.

not evaluates to True if its operand evaluates to False. It evaluates to False if its operand evalutes to True.

What do you think the following expression evaluates to? Try it out in the Python interpreter.

```
>>> True and not False or not True and False
```

It is difficult to read complex expressions, like the one above, and understand how a program will behave. Using parentheses can make your code easier to understand. Just so you know, Python interprets that expression in the following way:

```
>>> (True and (not False)) or ((not True) and False)
```

This is because boolean operators, like arithmetic operators, have an order of operation:

not has the highest priority
and
or has the lowest priority

It turns out and and or work on more than just booleans (True, False). Python values such as 0, None, ” (the empty string), and [] (the empty list) are considered false values. All other values are considered true values.

Question: What Would Python Display? (WWPD)

```
>>> not 2


-----------
>>> not not 2


-----------
>>> 2 < 3 or 4 > 5


-----------
>>> 1 < 2 and 3 > 2


-----------
```

Short Circuiting

What do you think will happen if we type the following into Python?

```
>>> 1/0
```

Try it out in Python! You should see a ZeroDivisionError. But what about this expression?

```
>>> True or 1/0
```

It evaluates to True because Python's and and or operators short-circuit. That is, they don't necessarily evaluate every operand.

Operator	**Checks if:**	**Evaluates from left to right up to:**	**Example**
AND	All values are true	The first false value	False and 1/0 evaluates to False
OR	At least one value is true	The first true value	True or 1/0 evaluates to True

Short-circuiting happens when the operator reaches an operand that allows them to make a conclusion about the expression. For example, and will short-circuit as soon as it reaches the first false value because it then knows that not all the values are true.

If and and or do not short-circuit, they just return the last value. Keep in mind that and and or don't always return booleans when using values other than True and False.

Error Messages

By now, you've probably seen a couple of error messages. They might look intimidating, but error messages are very helpful for debugging code. The following are some common types of errors:

Error Types	**Descriptions**
SyntaxError	Contained improper syntax (e.g. missing a colon after an if statement or forgetting to close parentheses/quotes)
IndentationError	Contained improper indentation (e.g. inconsistent indentation of a function body)
TypeError	Attempted operation on incompatible types (e.g. trying to add a function and a number) or called function with the wrong number of arguments
ZeroDivisionError	Attempted division by zero

Using these descriptions of error messages, you should be able to get a better idea of what went wrong with your code. **If you run into error messages, try to identify the problem before asking for help.** You can often Google unfamiliar error messages to see if others have made similar mistakes to help you debug.

For example:

```
>>> 2 + [1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

Identify what this error means.

2 Fundamentals of Functions

Defining Functions

Just like how we perceive functions in mathematics, all functions have an input, procedure, and an output. For example, a function, $f(x) = x^2 + 2x + 1$, is a quadratic function.

Three simple steps in computing this function. Let's say we **call** this function on 1.
Input: 1
Procedure: $1^2 + 2 * 1 + 1 = 4$
Output: 4

Let's try defining this function in Python:

```
>>> def namedoesntmatter(n):
...         return n**2 + 2*n + 1
...
>>> namedoesntmatter
<function namedoesntmatter at 0x100561e18>
>>> namedoesntmatter(1)
4
```

'def' shows that we will be defining a function.
'namedoesntmatter' would be the name of the function (a terrible name tho).
'(n)' is the parameter(input) of the function. Parameters could be more than one.
A function is an object that is available for taking in some (or no) parameters.
A function call is where you've actually take in an input and apply the function to the input.

We will learn how to construct the procedure of a function in the later sections.

Question: WWPD?

```
>>> def f(n):
... return n**3 + n + 5
...
>>> f(2)

-----------
>>> f(3) + 1

-----------
>>> def f(n, k):
... return n + k
...
>>> f(1, 5)

-----------
>>> f(4)

-----------
```

return and print

Most functions that you define will contain a return statement. The return statement will give the result of some computation back to the caller of the function and exit the function. For example, the function square below takes in a number x and returns its square.

```
def square(x):
    """
    >>> square(4)
    16
    """
    return x * x
```

When Python executes a return statement, the function terminates immediately. If Python reaches the end of the function body without executing a return statement, it will automatically return None.

In contrast, the print function is used to display values in the Terminal. This can lead to some confusion between print and return because calling a function in the Python interpreter will print out the function's return value.

However, unlike a return statement, when Python evaluates a print expression, the function does not terminate immediately.

```
def what_prints():
    print('Hello World!')
    return 'Exiting this function.'
    print('61A is awesome!')

>>> what_prints()
Hello World!
'Exiting this function.'
```

Notice also that print will display text without the quotes, but return will preserve the quotes.

Question: Pythagorean Theorem

```
def pyth(x, y):
    """Returns a square of hypotenuse of a right triangle,
    if x and y were to be adjacent and opposite respectively
    >>> pyth(3, 4)
    The hypotenuse squared is...
    25
    >>> pyth(1, 2)
    The hypotenuse squared is...
    5
    """
    print(___________________________)

    return__________________________
```

Conditional Statements

A conditional statement in Python consists of a series of headers and suites: a required if clause, an optional sequence of elif clauses, and finally an optional else clause:

```
if <expression>:
    <suite>
elif <expression>:
    <suite>
else:
    <suite>
```

When executing a conditional statement, each clause is considered in order. The computational process of executing a conditional clause follows.

1. Evaluate the header's expression.
2. If it is a true value, execute the suite. Then, skip over all subsequent clauses in the conditional statement.

If the else clause is reached (which only happens if all if and elif expressions evaluate to false values), its suite is executed.

Python has a built-in function for computing absolute values.

```
>>> abs(-2)
2
```

Let's try defining this function.

```
>>> def absolute(n):
... if n < 0:
...     return -n
... else:
...     return n
...
>>> absolute(-2)
2
```

Question: Is it greater than 3?

```
def threegreat(n):
    """Print a statement, "It's greater!", if n is greater than 3, or
    print a statement, "It's less...", if n is less than or equal to 3.
    >>> threegreat(4)
    It's greater!
    >>> threegreat(3)
    It's less...
    """
    "***YOUR CODE HERE***"
```

While Loops

In addition to selecting which statements to execute, control statements are used to express repetition. If each line of code we wrote were only executed once, programming would be a very unproductive exercise. Only through repeated execution of statements do we unlock the full potential of computers. We have already seen one form of repetition: a function can be applied many times, although it is only defined once. Iterative control structures are another mechanism for executing the same statements many times.

Consider a sequence of increasing number from 1 to 10:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Each value is constructed by incrementing one from the previous step. We start with 1, and then we will be incrementing 1 every time we go! Take a look at the following example.

```
>>> def one_to_n(n):
...     i = 1
...     while i <= n:
...         print(i)
...         i += 1 #same as i = i + 1
...
>>> one_to_n(5)
1
2
3
4
5
```

Question: Only print the even integers!

```
def even_one_to_n(n):
    """Prints out the even numbers ranging from 1 to n
    >>> even_one_to_n(8)
    2
    4
    6
    8
    """
    "***YOUR CODE HERE***"
```

Higher Order Functions

A higher order function is a function that manipulates other functions by taking in functions as arguments, returning a function, or both. We will be exploring many applications of higher order functions. Let's take a look at an example.

```
>>> def f(x):
...     def g(y):
...         return x + y
...     return g
...
>>> add_one = f(1)
>>> add_one(2)
3
>>> add_two = f(2)
>>> three = add_two(1)
>>> three + 1
4
>>> f(3)(4)
7
```

Here, when we call the function f on 1, it would return the inner function g with an information that x is 1. If we were to call the returned function to another integer, it would add that integer to 1 and return that. The Important catch is that functions can be defined within functions, and can be utilized within it. Make sure to check what you are returning and where you are at. Keep in track of all the parameters and functions. If you are curious of the frames, use pythontutor online.

Question: Make a Higher-Higher-Order function that adds three numbers!

```
def f(x):
    """Returns a higher-order function that would add three numbers!
    >>> f(2)(3)(4)
    9
    >>> a = f(2)
    >>> a(3)(5)
    10
    >>> a = f(1)
    >>> b = a(5)
    >>> b(6)
    12
    """
```

Lambda Funcitons

Lambda expressions are one-line functions that specify two things: the parameters and the return value.

```
lambda <parameters>: <return value>
```

While both lambda and def statements are related to functions, there are some differences.

	lambda	**def**
Type	lambda is an expression	def is a statement
Description	Evaluating a lambda expression does not create or modify any variables. Lambda expressions just create new function objects without changing the current environment.	Executing a def statement will create a new function object and bind it to a variable in the current environment.
Example	lambda x: x * x	def square(x): return x * x

A lambda expression by itself is not very interesting. As with any values such as numbers, Booleans, and strings, we usually:

- assign lambdas to variables (foo = lambda x: x)
- pass them in to other functions (bar(lambda x: x))

Try understanding the following code, write what you think.

```
>>> (lambda x: x * x)(2)
4
>>> addone = lambda y: y + 1
>>> addone(2)
3
>>> add_two_numbers = lambda x, y: x + y
>>> add_two_numbers(2, 3)
5
>>> weird_add = lambda a: lambda b: a + b
>>> weird_add(4)(5)
9
```

Question: Write the lambda versions for the following code!

```
def f(x, y, z):
    return (x+y)//z


def f(a):
    def g(b):
        def h(c):
            return a + b + c
        return h
    return g
```

3 Citation and Sources

Example questions and explanations are brought from the following sources:

1. Hilfinger Paul, DeNero John, “CS 61A: Structure and Interpretation of Computer Programs.” CS 61A Fall 2017, cs61a.org.

2. Point, Tutorials. “Python Numbers.” www.tutorialspoint.com, Tutorials Point, 5 Oct. 2017, www.tutorialspoint.com/python/python_numbers.htm.