# Isaac Computer Science Student Activity Booklet

**The Functional Programmer's Crash Course: Immutable & Elegant**

**Activity 1: Domain & Co-domain Trinket -** [**http://tiny.cc/er9uvz**](http://tiny.cc/er9uvz)

**A screenshot of a computer

Description automatically generated**

* If a student receives a score of 105, is that within our function's domain?"

105 is not within the function's specified domain. The domain is from 0 to 100 inclusive.

* If you call the *grade\_student()* function passing in each of the *test\_scores in turn .* “D” and “F” will not be returned, are these grades still in the co-domain?

Yes, "D" and "F" are still part of the co-domain as they represent possible outputs of the function. The co-domain is a set of all potential outputs, not just those that result from the current set of test scores.

Which scores would output i) “D” ii) “F”?  
  
i) A score that outputs "D" would be any score between 50 and 59, inclusive.

ii) A score that outputs "F" would be any score below 50.

**Activity 2a: First class objects** **Trinket:** [**http://tiny.cc/pnauvz**](http://tiny.cc/pnauvz)

1. What is the result of assigning greet to say\_hello and then calling say\_hello()?

Assigning greet to say\_hello makes say\_hello a reference to the greet function. Therefore, calling say\_hello() invokes the greet function and will print "Hello!".

1. How does the compute function illustrate the concept that in Python, functions are first-class objects?"

The compute function illustrates that in Python, functions are first-class objects because it treats functions just like any other value. It accepts a function as an argument (func) and a value (value), and then calls the passed function with that value.

1. In the multiplier function, how is the factor variable being used by the nested multiply\_by\_factor function?  
     
   The nested function multiply\_by\_factor captures the factor variable from the multiplier function when it is returned. Despite multiplier finishing its execution, the multiply\_by\_factor function retains access to the factor variable. This allows the multiply\_by\_factor function to remember the state of its environment when it was created. When double is called with the argument 5, it uses the captured factor value which is 2, multiplying 5 by 2 to produce 10.

**Activity 2b: Data structures Trinket:** [**http://tiny.cc/qvgyvz**](http://tiny.cc/qvgyvz)

**Predict:**

* Before running the program, predict what the program does. Can you guess what the math\_quiz function's purpose is and how it uses the operations list?

**Run:**

* Run the program and observe the output. Does it match your predictions?

**Investigate:**

* Look at how the operations list is structured. It contains tuples, each with a string representing an arithmetic operation and a corresponding function object.
* Manually select a tuple from the operations list and call the function yourselves with two numbers and see the result.

**Modify:**

* Add another operation, like modulo (%), to the operations list. You will need to write a new function and add it to the list to see if it gets selected and executed during the quiz.

def modulo(x, y):

return x % y # New modulo function

# Store functions in a list

operations = [

("+", add),

("-", subtract),

("\*", multiply),

("/", divide),

("%", modulo) # Add the modulo operation to the list

]

**Make:**

* Finally, try and make your own version of the program. You could create a more complex quiz, perhaps with different types of questions, or you could store the functions in a different data structure, like a dictionary, to see how it changes the way the functions are selected and executed.

# Use a dictionary to store functions

operations\_dict = {

'+': add,

'-': subtract,

'\*': multiply,

'/': divide,

'%': modulo

}

# Randomly select an operation from the dictionary keys

symbol = random.choice(list(operations\_dict.keys()))

operation = operations\_dict[symbol]

**Activity 3: Partial functions**  **Trinket:** [**http://tiny.cc/5jgyvz**](http://tiny.cc/5jgyvz)

* Modify the Trinket program to include a "buy one, get one half off" discount function. This discount applies 50% off the price of a second item of equal or lesser value when two items are purchased. You will need to create a new function that calculates the discount and then use partial to create a specialized function for it.
* Follow these steps to complete the task:
  1. Define a new function named bogo\_half\_off that accepts two arguments: price1 and price2. This function should calculate the total cost when the second item is half-price.
  2. Use functools.partial to create a new specialised bogo function with the price of the first item. This new function should only need one price argument to calculate the total cost. Assume, that the second price is less than or equal to the first.
  3. Add test cases to check the new bogo\_half\_off function with different price inputs.

### # New BOGO discount function def bogo\_half\_off(price1, price2):

### return price1 + (price2 / 2)

### original\_price = 100

### # Create a specialized BOGO function bogo\_discount = partial(bogo\_half\_off, original\_price) # Testing

### second\_item\_price = 50 print(f"Price for two items with BOGO discount: £{bogo\_discount(second\_item\_price)}")

**Activity 4:** **Image Processing Trinket:** [**http://tiny.cc/2hiyvz**](http://tiny.cc/2hiyvz)

* Use partial from the functools module to create new functions from the existing ones. For example, create a new function that always applies a blur with a specific strength or reduces the size to a specific percentage of the original.
* Refactor the process\_image function to use these new partial functions instead of the original functions.
* Test the refactored process\_image function to verify that it still performs the same sequence of operations correctly.

# Refactor the process\_image function

def reduce\_size(img, factor = 1):

width, height = img.size

new\_width = int(width \* factor)

new\_height = int(height \* factor)

return img.resize((new\_width, new\_height))

# Create a specialized function that always doubles the image size

double\_size = partial(reduce\_size, factor=2)

# Now, to compose the functions:

def process\_image(img):

return double\_size(apply\_blur(to\_grayscale(img)))

**Activity 5a: Immutable data Trinket:** [**http://tiny.cc/h2kyvz**](http://tiny.cc/h2kyvz)

**Challenge 1:**

* Identify the different 4 data types shown in lines 5 to 8 of the Trinket code.

list ['a', 'b', 'c']

tuple ('a', 'b', 'c')

string ‘abc’

integer 123

* Uncomment each of the data types one at a time and attempt to change one of the elements to observe what happens.
* Why does Python raise a TypeError when trying to change an integer, tuple or string, but not a list.

The list is mutable, which means its elements can be changed. The tuple string and integer are immutable, which means once they are created, their elements cannot be changed.

**Challenge 2:**

* Predict what will happen when each data type is used as the key for my\_dictionary. Which ones will cause a TypeError and why?

Integer 123 - Will not cause an error because integers are immutable.

String '123' - Will not cause an error because strings are immutable.

Tuple (1, 2, 3) - No error if all elements in the tuple are also immutable.

List [1,2,3] - Will cause a TypeError because lists are mutable.

* Uncomment each data type one at a time (integer, string, tuple, list) and attempt to run the code.
* For each TypeError explain why it was raised.

If you uncomment each data type one at a time and use it as a key in **my\_dictionary**, you will find that the integer, string, and tuple can be used as keys without any issues. However, when you uncomment the list and attempt to use it as a key, Python will raise a **TypeError** indicating that lists cannot be used as dictionary keys because they are mutable.

* Why is it important that dictionary keys are immutable?

After inserting a key-value pair into the dictionary, changing the data in the key would effectively "lose" the entry because the dictionary's hash table depends on the key's hash value being stable. The dictionary wouldn't be able to find the original entry using this modified key.

**Activity 5b: Pure functions & side effects Trinket:** [**http://tiny.cc/0glyvz**](http://tiny.cc/0glyvz)

* Predict the behaviour of each function and decide whether it is pure or impure. Base your predictions on whether the function's return value is solely determined by its input parameters and whether the function has any observable side effects.

**square(x):** This function is pure. It does not modify any external state and its output is solely determined by its input parameter **x**.

**increment\_and\_square(x):** This function is impure because it modifies the global variable call\_count, causing an observable side effect.

**save\_to\_file(data, filename):** This function is impure as it performs I/O operations by writing to a file, which is an external side effect.

**add\_to\_shared\_data(value)**:This function is impure because it modifies the external shared\_data list.

* Call each function with specific arguments and observe any changes outside the function's scope. You should call increment\_and\_square multiple times with the same argument to see if the global call\_count changes. You should also call add\_to\_shared\_data to see if the shared\_data list is modified.

After calling these functions, students will observe that the global variable **call\_count** changes and the **shared\_data** list is modified, confirming their impurity.

* After you have investigated each function, make a final decision which are pure and which are impure. Record the side effects, that were observed and why they matter.
* Modify the impure functions to make them pure, if possible. For example, you might remove the global variable and pass **call\_count** as a parameter, return a new list instead of modifying **shared\_data.**

def increment\_and\_square(x, call\_count):

call\_count += 1

return x \* x, call\_count

def add\_to\_shared\_data(value, shared\_data):

new\_data = shared\_data + [value]

return new\_data

**Activity 6: Higher order functions Trinket:** [**http://tiny.cc/wstyvz**](http://tiny.cc/wstyvz)

* Predict what each higher-order function does with the provided function argument. For example, what does map(square, numbers) do with the square function and the numbers list?

**map(square, numbers)** applies the **square** function to each element of the **numbers** list, returning a new list where each number is squared.

**filter(is\_even, numbers)** goes through each element of the **numbers** list and applies the **is\_even** function to determine if the number is even. It returns an iterable of only those numbers for which **is\_even** returns **True**.

**reduce(multiply, numbers)** applies the **multiply** function cumulatively to the items of the **numbers** list, from left to right, so as to reduce the list to a single value. This would effectively calculate the product of all the numbers in the list.

* Run the program and observe the output. Does it match your predictions? Change the arguments and note the effects.
* Modify the functions passed to map, filter, and reduce. For example, you could write a new function to replace square that adds a constant to each number and observe how that affects the output of map; you could use **filter** to extract numbers from a list that are prime or for **reduce**, you could change the operation to addition.

def add\_constant(x):

return x + 10

added\_numbers = map(add\_constant, numbers)

print(list(added\_numbers)) # Output: [11, 12, 13, 14, 15, 16]

def is\_prime(x):

if x < 2:

return False

for n in range(2, x):

if x % n == 0:

return False

return True

prime\_numbers = filter(is\_prime, numbers)

print(list(prime\_numbers)) # Output: [2, 3, 5]

def add(x,y):

return x + y

sum\_numbers = reduce(add, numbers)

print(sum\_numbers) # Output: 21

**Activity 7: Head & tail operations Trinket:** [**http://tiny.cc/j2wyvz**](http://tiny.cc/j2wyvz)

* Predict what the head and tail will be for the sample\_list and what the output of both recursive functions will be when called with numbers.

The head of sample\_list is 5, and the tail is **[8, 2, 9, 7]**.

For the recursive\_sum function, it will sum all the numbers recursively and return 15.

For the recursive\_reverse function, called with the same list, it will reverse the list recursively and return **[5, 4, 3, 2, 1]**

* Run the code and verify if your predictions are correct.
* Change the sample\_list and numbers list and observe how the head and tail change and how the output of the recursive functions changes.
* Modify the recursive\_sum use lst[-1] as the head and lst[:-1] as the tail. What impact does this have on the output?

If you modify recursive\_sum to use lst[-1] as the head (the last element of the list) and lst[:-1] as the tail (all elements except the last), the function will still correctly calculate the sum of the list. The order in which you add numbers does not affect the sum.

* Make a new recursive function that uses head and tail operations to perform a different task, such as finding the maximum value in a list.

def recursive\_max(lst):

# Base case: if list is empty, return a value that won't affect the max

if not lst:

return float('-inf')

# If there's only one element, it's the max

if len(lst) == 1:

return lst[0]

# Recursive case: max is the greater of the head and the max of the tail

return max(lst[0], recursive\_max(lst[1:]))

# Test the function

print(f"Maximum of {numbers} is {recursive\_max(numbers)}")

You can use 0 instead of **float('-inf')** as a base case, with one caveat: using 0 assumes that all the numbers in the list are positive. If the list could contain negative numbers, using 0 could incorrectly identify 0 as the maximum when, in fact, all numbers in the list are negative and less than 0.

**Activity 8: Procedural vs functional Trinket:** [**http://tiny.cc/pxwyvz**](http://tiny.cc/pxwyvz)

* Review the Trinket code and note the differences in how the tasks of doubling numbers and filtering even numbers are achieved in procedural vs. functional programming.

In the procedural style, doubling numbers is done by iterating over the list with a loop and replacing each element with its double. Filtering even numbers involves another loop where each number is checked for evenness, and if it passes, it is added to a new list of even numbers.

In the functional style, the doubling would be done using the **map** function, which applies a given function (in this case, a doubling function) to each item of a list. Filtering even numbers would use the filter function, which applies a boolean function (in this case, checking for evenness) to each item of a list and returns a list of items where the function returned True.

* Before running the code, predict the output of both code blocks.
* Identify potential advantages and disadvantages of each approach.

**Procedural Programming:**

Advantages:

* + Can be more intuitive as it follows a clear sequence of steps.

Disadvantages:

* Can lead to more complex code with side effects that make it harder to debug.
* Less concise compared to functional programming.
* May not take advantage of modern multi-core processors as effectively.

**Functional Programming:**

Advantages:

* Leads to more concise and readable code.
* Functions without side effects (pure functions) are easier to debug and test.
* Functional code is often more scalable and can be parallelized more easily.

Disadvantages:

* Can be less intuitive for those who are not familiar with the functional paradigm.
* Sometimes functional code can be less efficient in terms of memory and speed due to the use of immutable data structures and recursion.