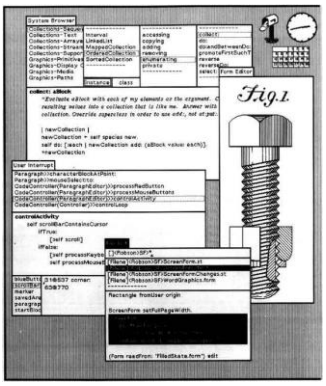
Code by Numbers



Is your computer running slow? Time to upgrade? Think again; computer upgrades are not what they used to be. Improvements in computing speed traditionally came from CPU and memory upgrades; most improvements now come from improved algorithms and software developments. Graphical user interfaces demand more power and memory. To understand what is happening, we need to step back a few decades.

In 1983 Apple® introduced LISA, the first *commercially* available computer with a graphical user interface (GUI), the 5Mb hard drive was an optional extra. The inspiration for Lisa came from something Xerox® had been using for almost a decade. The photocopying behemoth had created a computer called: "Alto" to help make their offices run more efficiently. The GUI that Xerox developed (1974) was called Smalltalk. Users had a three-button mouse, desktop clock, calendar, email, spreadsheet and a word-processer. It is fair to say Xerox was creating the paperless office, not exactly something a photocopying company would want widely adopted. Ironically, they also developed the laser printer, something that would become more pervasive through the use of personal computers.

Smalltalk was the first object-oriented language allowing programmers to use these objects within their programs without having to know *how* the object worked. In 1985 Microsoft® introduced Windows®. Applications in Windows ran on a platform called MS.DOS (Microsoft's Disk Operating System). Creating an application to work in Windows meant you could rely on MS.DOS to handle all the background computer management. Windows 1.0 required 256kB of memory and at least two floppy drives or 256kB hard-drive. The processor speed was approximately 5MHz and only 16bit. Fast-forward 35 years, Windows 11 requires a minimum 1 GHz twin core 64bit processor, 4GB RAM and 64GB storage. How did we get to the point? The simple answer is that processors and memory became cheaper allowing developers to add more and more layers to their code. From the 1980's to early 2000's, computer processing speeds increased almost exponentially. For the past 15 years, the same speed increases have not been possible.

This booklet aims to illustrate how mathematics and coding can be used to improve computational speeds without relying on hardware updates. The activities and investigations are also designed to improve understanding of number. The hierarchical nature of the mathematical content and structure of the coding require these activities be completed sequentially. Each activity includes coding instructions and references, visual and instructional support for mathematical content, reflective questions and a detailed investigation. The 10 minutes of coding references should be completed before commencing the corresponding activity. Coding instructions are included in each activity; however, it is also recommended that the code be modified to improve performance. The first activity "Factors that Count" involves writing a program to count the quantity of factors for any given number. The sample code provided is a 'brute force' approach; this code can be modified to achieve the same result much, much faster! The "Euler Totient" activity repeatedly requires information about the factors of a number; the factor program could be called upon for this purpose, however alternative algorithms can also be used that make the program run many, many times faster.

Why focus on number? Aside from the fact that the mathematical content is accessible, the importance of factors, or the lack of them (prime numbers) is critical to our world's economy thanks to encryption. Many trillions of dollars are digitally transacted every day, the security of these transfers relies on prime numbers and the fact that current algorithms are not particularly efficient at *disassembling* numbers.

Author: P. Fox

Table of Contents

Factors that Count 3
Finding and Counting Factors 3
Writing a Program 4
Investigation 6
Euclid's Algorithm 7
Highest Common Factors 7
Writing a Program 8
Investigation 9
Euler Totient Function 10
Introduction 10
Writing a Program 10
Investigation 12
Highly Composite Numbers 13
Introduction 13
Writing a Program 13
Investigation 15

Author: P.Fox

Factors that Count

TI-Codes Lessons:

Unit 1 – Skill Builder 1

⇩

Unit 4 – Skill Builder 1

Commands:

- input
- for (range)
- if
- print
- int (number types)
- [ ] (create a list)
- Append (add elements to a list)
- len (length of a list)
- % (modular arithmetic)

Finding and Counting Factors

There are many ways to determine the quantity of factors for a specified number. The most common method is to test the divisibility for all applicable numbers. For example, suppose we want to determine the quantity of factors for 18. We can determine the quotient and remainder for all the numbers from 1 to 18.

Table 1A – Finding the factors of 18

Divisor	1	2	3	4	5	6	7	8	9
Quotient	18	9	6	4	3	3	2	2	2
Remainder	0	0	0	2	3	0	4	2	0

Table 1B – Finding the factors of 18

Divisor	10	11	12	13	14	15	16	17	18
Quotient	1	1	1	1	1	1	1	1	1
Remainder	8	7	6	5	4	3	2	1	0

Our conclusion is that 18 has six factors since there are six occasions whereby the remainder is equal to zero.

This divisibility check for all numbers is exhaustive. You may have ideas about how this process can be made more efficient, however, this method will provide a basis for an algorithm on which to write a simple program to count the quantity of factors for a given number. You can make the necessary improvements and checks once your initial program is complete and functioning.

Question: 1.

Write a description of the steps required to determine the quantity of factors for any whole number: n.

Instructions:

Start a new document and insert a calculator application.

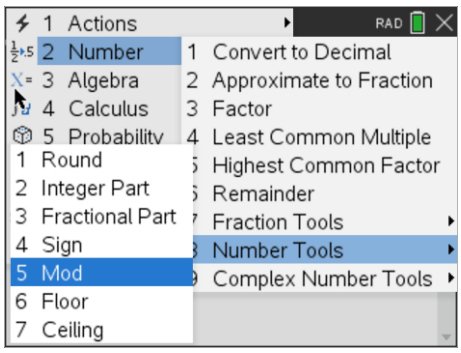
Locate the **mod** command using: **Number** > **Number Tools** > **Mod**

Determine the result of the following calculations:

Mod(18,6)

Mod(18,5)

Mod(18,12)



Question: 2.

Based on your experimentation, what value does the MOD command return?

Question: 3.

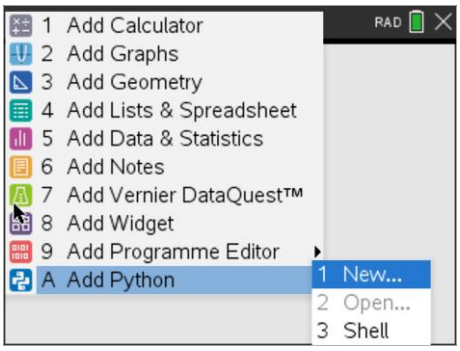
If MOD(*a*, *b*) = 0, what does this say about the relationship between *a* and *b*?

Author: P.Fox

Writing a Program

Create a new Python program by pressing:

ctrl + doc , **Add Python > New…**



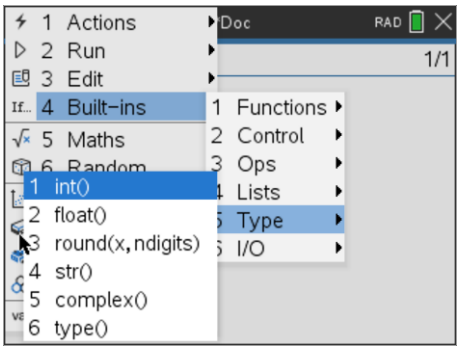
Call the program: FactorCount

Note that 'FactorCount' is one word as program names cannot contain spaces.

If the programming application is launched on the same page as the Calculator Application. The page-layout in the document menu can be used to give each application its own page.
Short-cut: [**Ctrl**] + [**6**]. Page 1.1 = Calculator Application. Page 1.2 = Program Application.

The first task is to request a number from the program user and store it as a variable. Type in:

n =



Python input defaults to text, so the next step is to restrict the input to a whole number (integer). The int() command is located in the "**type**" menu, alternatively it can be typed in directly from the keyboard:

Press:

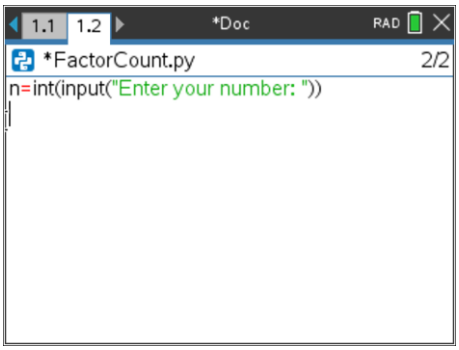
menu > **Built-ins > Type > int()**

The next step is to enter the "input" command:

menu > **Built-ins > I/O > input()**

Finish the instruction by adding a text prompt.

When this command is executed, the user's numerical input will be stored in a variable "n"



- Quotation marks: " " can be entered by pressing [**Ctrl**] + [**x**] (multiplication sign)
- Colour is added automatically to help locate the various parts of the syntax.

We could just count factors, however, it is easy record and store them in a list which also helps with checking the results.

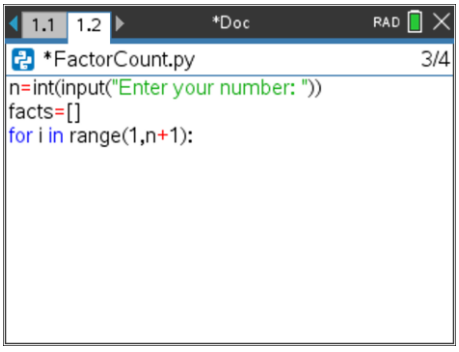
Factors = [ ]

This creates an empty list called factors.

A "FOR" loop can be used to check for factors. We use a FOR loop because we can pre-determine the quantity of iterations the loop must perform.

menu > **Built-ins > Control > For index in Range(***size***)**

Python loop execution ceases when the counter reaches *size*, therefore the counter (*size*) needs to be one more than n.



Author: P.Fox

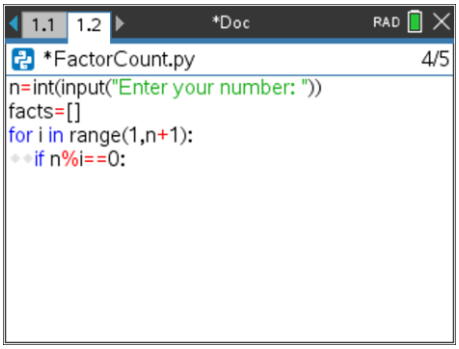
An **IF** statement will be used to check if the user's number has a factor each time the program executes the loop.

The IF command can be selected by:

menu > **Built-ins** > **Control** > **if**

The % operation in Python is for modular arithmetic.

The percentage sign can be obtained from the punctuation fly-out menu.

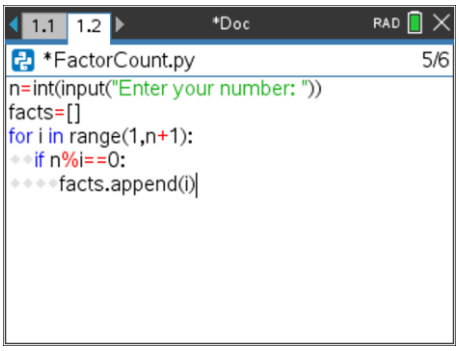


The 'append' command adds the latest factor to the current list of factors. This command can be typed in directly or accessed from the variable menu (facts) followed by:

menu > **Built-ins** > **Lists** > **.append()**

The value to be added to the list, given that the division has not generated any remainder, is "*i*", the loop counter.

That's all for the algorithmic part of the program! The next step is to display the results. Start by deleting the indentions, the end of the "IF" condition and the For loop.



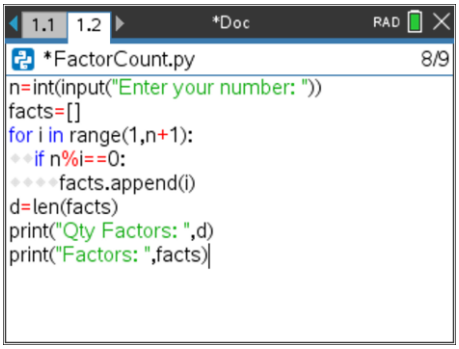
The list ("facts") contains all the factors, len(facts) therefore returns the size of the list. This quantity can be stored in 'd'.

Now the quantity of factors (d) and the actual factors can be printed to the screen.

Save the program and launch it by pressing [**Ctrl**] + [**R**]. A new Python shell will be created and the program name automatically pasted. Start by checking the factor count for 18.

The table at the start of this activity identifies 6 factors, compare this with the output from your program.

To check another number, press [**Ctrl**] + [**R**]



Question: 4.

Determine the quantity of factors for each of the following numbers:

a. 24

b. 36

c. 37

d. 144

Check each of your answers by writing down all the factors.

Question: 5.

Determine the quantity of factors for each of the following numbers. Identify a specific characteristic about the quantity of factors and use this to classify the numbers into two groups, explain your classification.

29, 84, 104, 87, 22, 37, 101, 97, 45, 43, 133, 153, 173, 107

Question: 6.

Determine the quantity of factors for each of the following numbers. Identify a specific characteristic about the quantity of factors and use this to classify the numbers into two groups, explain your classification.

28, 30, 90, 45, 50, 60, 120, 72, 25, 49, 81, 144, 441, 82, 24, 720.

Author: P.Fox

Question: 7.

The FactorCount program works, but it could be more efficient. Use a stop watch to time how long the program takes to count the number of factors for: 100,000; 200,000 and 300,000. Use these times to predict how long the program will take to count the factors for 500,000. Test your answer!

If you are satisfied with your prediction, how long would it take to find factors for the following number:

2,140,324,650,240,744,961,264,423,072,839,333,563,008,614,715,144,755,017,797,754,920,881,418,023,447,140,136,643,345,519,095,804,679,610,992,851,872,470,914,587,687,396,261,921,557,363,047,454,770,520,805,119,056,493,106,687,691,590,019,759,405,693,457,452,230,589,325,976,697,471,681,738,069,364,894,699,871,578,494,975,937,497,937 [250 digits!]

Note: This number is associated with RSA Encryption.

Investigation

Why are factors important? To answer this question, consider the opposite situation, the *absence* of factors. Billions of dollars are moved around electronically every day, to do this securely, the electronic transfers must be encrypted. The most common encryption method (RSA) is built around very large prime numbers, numbers with an *absence* of factors! All encryption methods are essentially built on numbers, so being able to 'assemble' and 'disassemble' numbers is extremely important.

Your task is to find a rule that determines the quantity of factors for any whole number, given the prime factorisation of that number.

A few clues are provided along the way to help you on your factor sleuth journey. Document your search findings and conclusions using the clues and your program to help expedite your investigation.

Clue 1:

Determine the prime factorisation and corresponding quantity of factors for the following numbers: 36; 100; 441; 3025 & 48841.

Clue 2:

Determine the prime factorisation and corresponding quantity of factors for the following numbers: 24; 250; 1029; 6655 and 198911.

Clue 3:

Based on the first two clues, generate some numbers that you believe have exactly 8 factors. Test your numbers and comment on the results.

Clue 4:

Determine the prime factorisation and corresponding quantity of factors for the following numbers: 2000; 64827; 107811; 668168 and 1585615607. [Note: For this last number you will need a fast algorithm!]

Clue 5:

Create some numbers of the form: $m^2 \times n^5$ where m and n are both prime. Determine the quantity of factors for each of your numbers. [Note: You may want to choose relatively small prime numbers for m and n.]

Continue the exploration, tabulate your results and record your thoughts, hypotheses, tests and reflections as you go. Documenting findings is an important part of the investigative process. Detectives may have many suspects in their initial investigations, however as more clues surface they develop hypotheses. Detectives test each hypothesis, review what they already know or go in search of more clues. Some investigations end up as Cold Cases, however it is critical that detailed documentation of all aspects of their investigation are retained in the event the investigation is re-opened. Some crimes remain unsolved despite having significant suspects, in mathematics these are often called 'conjectures', a theory that seems to work but has never been proven.

Author: P.Fox

Euclid's Algorithm

TI-Codes Lessons:

Unit 1 – Skill Builder 1

⇩

Unit 4 – Skill Builder 1

Commands:

- input
- while
- if
- else
- int (number types)
- print

Highest Common Factors

The Highest Common Factor (HCF) or Greatest Common Divisor (GCD) of two numbers is useful for many reasons. The process is valuable when working with fractions, solving packaging problems, developing traffic light sequences and encrypting content for digital communications. Developed more than 2000 years ago, Euclid's algorithm is still the most efficient process used to determine the Highest Common Factor of two numbers.

Euclid's Algorithm:

LINE #1: IF A = 0 THEN GCD(A,B) = B since GCD(0,B) = B

LINE #2: IF B = 0 THEN GCD(A,B) = A since GCD(A,0) = A

LINE #3: A = B x Q + R … where Q is the quotient and R is the remainder

LINE #4: GCD(B,R) = GCD(A,B), now find GCD(B,R)

This algorithm will make more sense when some numbers are used for A and B. Suppose we want to find the highest common factor of (A) 1260 and (B) 385. As neither A = 0 or B = 0 we progress to LINE #3.

1260 = 385 x 3 + 105 [We say that 105 is the remainder when 1260 is divided by 385]

According to LINE #4 of Euclid's algorithm: GCD(1260,385) = GCD(385,105)

We apply the algorithm again. Since 385 ≠ 0 and 105 ≠ 0 we proceed to LINE #3.

385 = 105 x 3 + 70 [We say that 70 is the remainder when 385 is divided by 105]

According to LINE #4 of Euclid's algorithm: GCD(1260,385) = GCD(385,105) = GCD(105,70).

We apply the algorithm again. Since 105 ≠ 0 and 70 ≠ 0, we proceed to LINE #3

105 = 70 x 1 + 35 [We can say that 35 is the remainder when 105 is divided by 70]

We are getting close! According to LINE #4 of Euclid's algorithm: GCD(1260,385) = … = GCD(70,35)

Applying the algorithm one more time, as 70 ≠ 0 and 35 ≠ 0, we proceed to LINE #3.

70 = 2 x 35 + 0. [This time the remainder is 0!]

Now we can apply LINE #1 or LINE #2 since we have GCD(35,0) = 35.

Our conclusion is that the Highest Common Factor or Greatest Common Divisor of 1260 and 385 is 35.

Question: 1.

Use Euclid's algorithm to identify the highest common factor of: 3850 and 3234.

Author: P.Fox

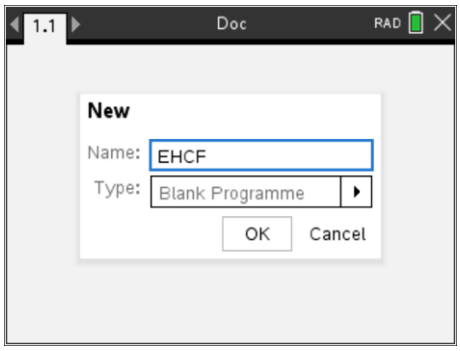
Writing a Program

Instructions:

Start a new document; insert a new Python program.

Add Python > New

Call the program: EHCF



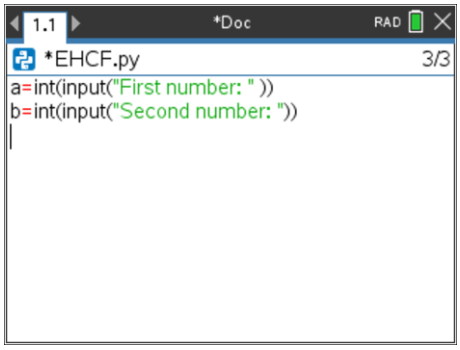
Insert two prompts for the integers 'a' and 'b'. The input type (integer) can be typed directly or entered via the menu.

Menu > Built-ins > Type

The input command can be typed directly or entered via the menu:

Menu > Built-ins > I/O

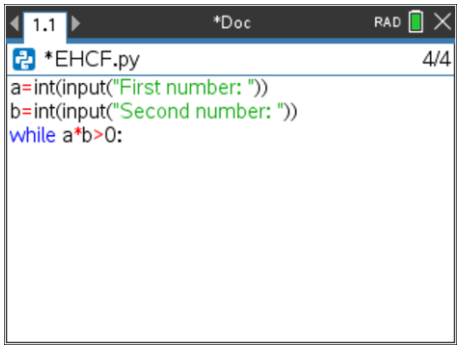
Include a text prompt for the user.



Euclid's algorithm ceases when either a = 0 or b = 0, an easy way to check this is: a x b = 0. The "null factor law" states that if the product of two numbers is zero, then one or both of the numbers must be zero.

The algorithm should continue to run while a x b $\neq$ 0.

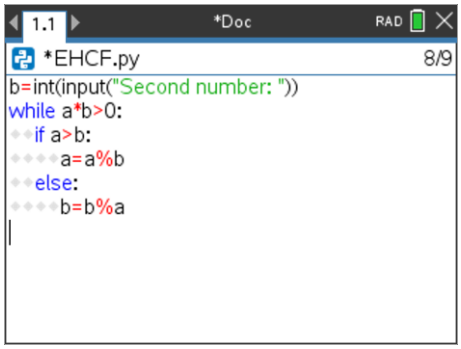
Menu > Built-ins > Control > while..



Modular arithmetic returns the remainder when a ÷ b (where a > b) so an if …else statement can be used to process Line #3 of Euclid's algorithm.

Menu > Built-ins > Control > if … else

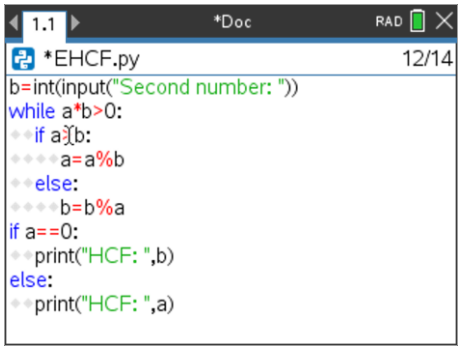
Enter the condition, then press Tab to navigate to the instruction block and use the % sign for modular arithmetic, then Tab to the second block to complete the instruction block.



That's the entire algorithm! The only thing remaining is to display the results. You could display as:

print(*a*,*b*)

Alternatively, use an if … else option to only display only the highest common factor rather than both a and b. (Shown opposite.)



Author: P.Fox

Question: 2.

Determine the highest common factor of: 1914 and 7293 (by hand) using Euclid's algorithm and use your results to check the program.

Question: 3.

Test your program on some smaller numbers where you know the highest common factor. Record your test results.

Question: 4.

The **Number** menu in the Calculator Application contains a command to determine the highest common factor of **two** numbers. Edit your program to find the highest common factor of three numbers.
Example: EGCD(a,b,c)

Test and evaluate your program.

Question: 5.

Edit your program to test for the highest common factor of an entire list of numbers.
Note: The program can be defined as egcd(data) where data is a list of numbers: {#, #, # ...}. The **dim** command can be used to determine the dimensions (quantity of numbers) entered into the list.

Investigation

The prime factorisation of a number can be used to efficiently find the highest common factor of any two or more numbers. Use your program to find the highest common factor for each list of numbers (below). Write the original numbers and the highest common factor in terms of their prime factorisation. Try some of your own lists, then write a description of how you can use the prime factorisation to determine the highest common factor of any two or more numbers.

List 1: 1260, 1410, 2040, 4290 & 9570

List 2: 220, 1400, 1700, 30940 & 154700

List 3: 2964, 3588, 8892, 10764 & 409032

List 4: 399, 441, 1911, 3381, 5733 & 835107

Euler Totient Function

TI-Codes Lessons:

Unit 1 – Skill Builder 1

⇩

Unit 4 – Skill Builder 1

Commands:

- input
- for (range)
- if
- print
- int (number types)
- def function
- while
- % (modular arithmetic)

Introduction

The Euler Totient Function for a whole number '*n*' counts the quantity of numbers that are co-prime up to the number *n*. To help understand this definition, consider the number 12.

We need to check which numbers: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12} have a factor in common with 12, these numbers are discarded leaving us with the numbers that are co-prime. This is summarised in the table below.

Whole Numbers < n	1	2	3	4	5	6	7	8	9	10	11	12
Highest Common Factor	1	2	3	4	1	6	1	4	3	2	1	12

There are 4 numbers where the highest common factor is 1, these numbers are co-prime with 12: {1, 5, 7, 11}. The Euler Totient function for 12 is therefore equal to 4, this can be written as: $\varphi(12) = 4$.

Here is another example for the number 9.

Whole Numbers < n	1	2	3	4	5	6	7	8	9
Highest Common Factor	1	1	3	1	1	3	1	1	9

The Euler Totient function for 9 is therefore equal to 6, this can be written as: $\varphi(9) = 6$.

Question: 1.

Create some pseudo-code for the Euler Totient function.

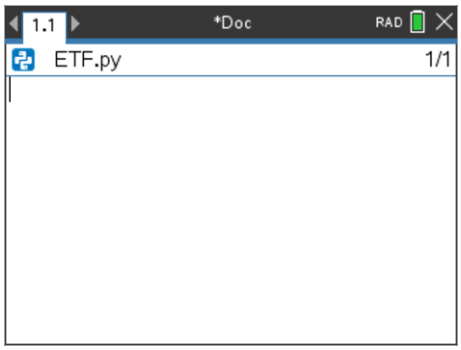
Writing a Program

Instructions:

Start a new document; insert a new Python program.

Add Python > New

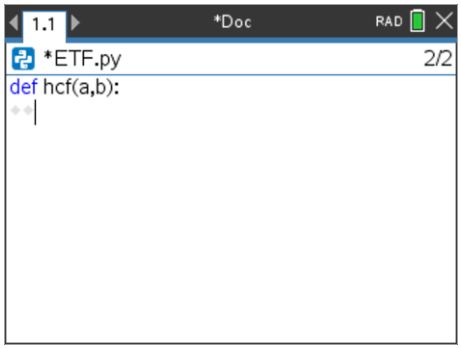
Call the program: ETF



The Euler Totient Function counts the quantity of numbers that are co-prime up to the specified number. When two numbers are co-prime their highest common factor is one, it therefore makes sense to use Euclid's algorithm to check the highest common factor. To do this efficiently, Euclid's algorithm can be defined as a function.

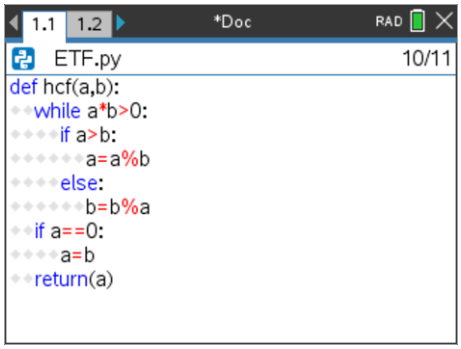
Built-ins > Functions > def function()

The function requires two parameters, the two numbers for which the highest common factor will be returned.



Author: P.Fox

Euclid's algorithm for the Highest Common Factor can now be deployed through this function as per the activity on Euclid's algorithm. The only difference here is at the end of the function: 'return(a)', this is the value returned once the function has been called.



Note:

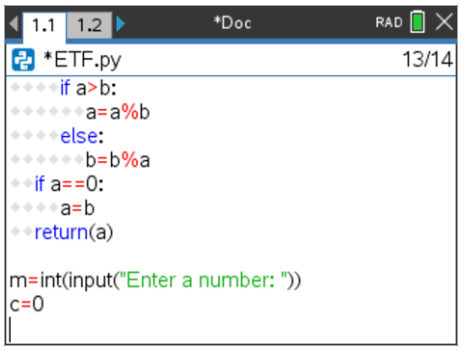
When the program runs, nothing happens with the function until it is called from the program.

The program needs to count the quantity of numbers that are co-prime with the selected (input) number. If two numbers are co-prime, their highest common factor is 1.

Start by requesting an input value and setting a counter equal to 0

m = int(input("Enter a number: ")

c = 0

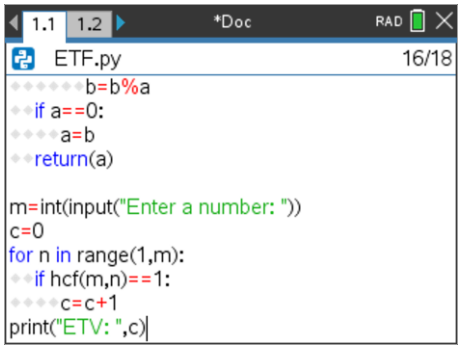


Note:

Variables 'a' and 'b' are used in the function, so it is best to avoid using them anywhere else in the program. Longer, more meaningful variable names can be used but keep them brief to avoid long lines of code.

The last part of the program is to scan the numbers from 1 to the designated value (m)* for numbers that are co-prime.

Each time one of these numbers is found, the counter increments by 1.



Note:

The loop will halt at 'm – 1', however the hcf(m,m)≠1 so this last check would not change the counter value c.

Question: 2.

Check that your program produces the same results for the two worked examples, then try several others (by hand) and compare results.

Question: 3.

Explore the Euler Totient function for prime numbers, what do you notice?

Question: 4.

Determine the fraction: $\frac{n}{\varphi(n)}$ for the following values of n: 30, 60 and 90, comment on the results.

Question: 5.

The number 100 can be expressed as: $2^2 \times 5^2$. Compare the Euler Totient value for 100 with the following calculation:

$$100\times\left(1-\frac{1}{2}\right)\left(1-\frac{1}{5}\right)$$

Author: P.Fox

Question: 6.

The number 1125 can be expressed as: $3^2 \times 5^3$. Compare the Euler Totient value for 1125 with the following calculation:

$$1125 \times \left(1-\frac{1}{3}\right)\left(1-\frac{1}{5}\right)$$

Question: 7.

Use the previous to questions to explore the prime factorisation approach to the Euler Totient function with the Euler Totient value determined by your program.

Question: 8.

How does the prime factorisation approach to calculating the Euler Totient function explain your results to Question 4?

Question: 9.

Why does the 'short cut' approach to the Euler Totient function work?

Investigation

Re-write the Euler Totient function program to determine the Euler Totient function for a range of numbers, graph the results and explore any patterns.

Note: Use the ti-system import module to share data from the program with the TI-Nspire document.

Author: P.Fox

Highly Composite Numbers

TI-Codes Lessons:

Unit 1 – Skill Builder 1

⇩

Unit 4 – Skill Builder 1

Commands:

- input
- for (range)
- if
- print
- int (number types)
- def function
- [ ] (create a list)
- Append (add elements to a list)
- % (modular arithmetic)
- Import module

Introduction

A highly composite number has more factors than any of its predecessors. Think of it as competition along the number line. The difficulty in locating highly composite numbers is that you must already know the previous highly composite number in order to identify how many factors the next number must have in order to qualify. Any search for highly composite number therefore generally starts at 1.

Whilst 1 only has one factor, there are no predecessors, so by default, 1 is the first highly composite number. Naturally 2 is the next highly composite number having two factors. The next is 4 with three factors then 6 with four factors. With one, two, three and four factors already checked, it would be easy to assume that the next highly composite number would have five factors, however 12 is the next highly composite number with six factors.

Question: 1.

Write a description of a program that will determine the Highly Composite number up to some value n.
Note: The quantity of factors for any number can be referenced as 'factor_count'.

Writing a Program

Instructions:

Start a new document; insert a new Python program.

Add Python > New

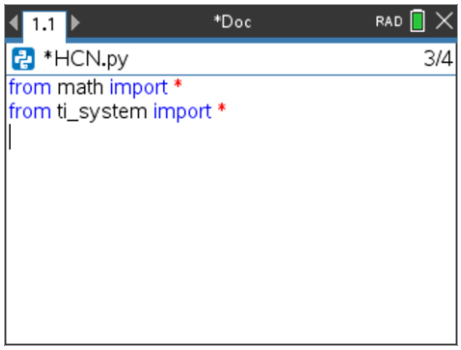
Call the program: HCN

To make the program efficient, it is desirable to have access to the 'square-root' function. Import the 'math' module.

Math > from math import

To access results outside the Python shell, import the TI-System module.

More Modules > TI-System > from ti-system import

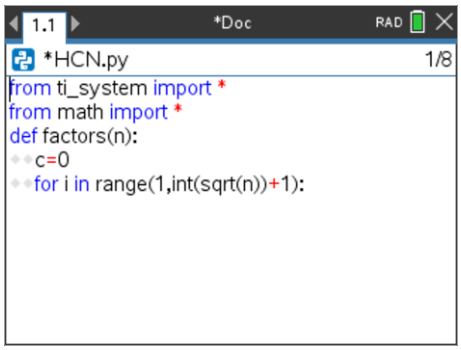


Creating a function to efficiently determine the quantity of factors will make the main program much easier.

Define a function called "factors" with input 'n':

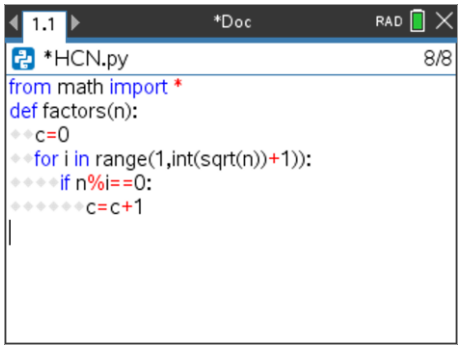
Built-ins > functions > def function()

A counter (c) will be used to count each factor with a loop to search for the factors. The loop only needs to go to the square-root of the chosen number, but a final check will be necessary in the event that the original number is a perfect square.



Author: P.Fox

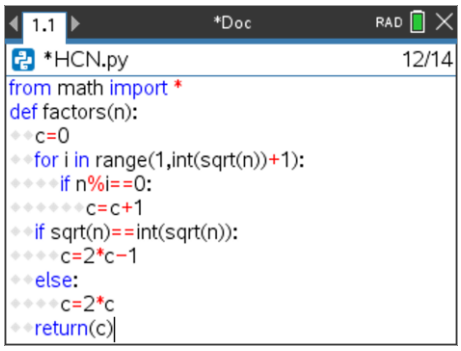
The loop checks if the current number (n) is divisible using modular arithmetic (%), if there is no remainder, then 'i' must be a factor of 'n', so the counter is increased by one.



Once the loop has finished, a check must be performed to see if the original number was a perfect square. If the original number was a perfect square, doubling the quantity of factors would count the square-root twice.

If the original number was not a perfect square, then the quantity of factors is doubled as all the factors counted to date have a 'partner'.

Finally, the quantity of factors (c) is returned to the program.

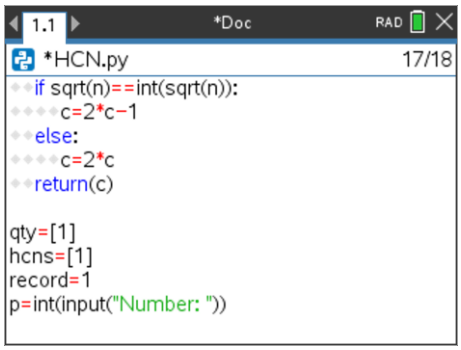


Several variables need to be initialised at the start of the program.

- QTY = The quantity of factors for the highly composite number
- HCNS = Highly Composite Numbers
- Record = Quantity of factors for the current HCN.

The first highly composite number '1' is seeded into the variables as it is the only 'odd' highly composite number.

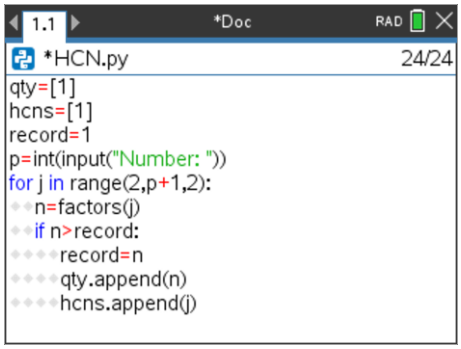
Note: "qty' and 'hcns' will hold a list of numbers that will be continually updated.



The loop can start at 2 since the first highly composite number (1) has already been stored. As all subsequent HCN's are even, the step counter can be set at 2.

The first instruction in the loop is to store the quantity of factors in variable 'n'; if this quantity is larger than the current record, the current record is updated and the 'qty' and 'hcns' lists are updated.

Note: The append command adds the specified value to the end of the specified list.



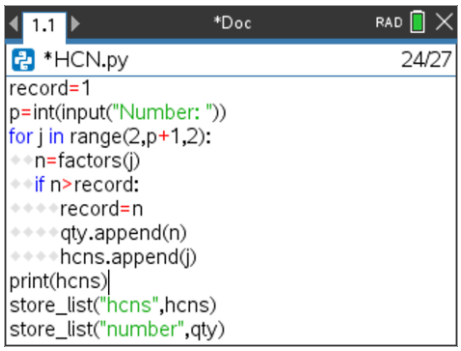
Once the loop is finished, all the highly composite numbers have been stored and can therefore be displayed and transferred to variables that can be accessed by the current document.

More Modules > TI System > store_list("name",list)

"name" represents the name of the variable in the current document.

"list" refers to the list in the current program (Python shell).

The program is now complete and ready to run.



Author: P.Fox

Question: 2.

Run your program and check that the first five highly composite numbers are: 1, 2, 4, 6, 12; then determine all the highly composite number from 1 to 100.

Question: 3.

Determine all the highly composite numbers from 1 to 1000 and their corresponding quantity of factors.

Question: 4.

Express each of the Highly Composite Number in the previous question as a product of its prime factors.

Question: 5.

Study the prime factorisations closely. Suggest a possible prime factorisation for the next highly composite number, the corresponding number and quantity of factors.
Note: You may have more than one educated guess.

Investigation

To continue exploring Highly Composite Numbers, a more efficient program (or new program) is required, one that no longer starts at 1, rather one that starts at some previously identified Highly Composite Number and uses information gleaned from the first sixteen highly composite numbers.

- Re-write your HCN program so that it can start at any HCN.
- Continue recording HCNs and the corresponding prime factorisations. When and what will be the next prime factor to be included in the prime factorisation?
- Identify any patterns you can find in the prime factorisation that would help in locating subsequent prime factorisations.
- What prior learning are you using to identify the quantity of factors, make predictions and search?