Intro to Computer Programmming

(With Fig and Python)

Why learn computing?

In an increasingly specialized world, we could be forgiven for asking, "aren't there people to do this stuff for us?" The answer is there are, and if not for those people we might better understand the tools we use on a daily basis.

Many of us live in an environment so advanced, it's as if there is a specialist for almost anything we can imagine.

Yet we celebrate each time a medicine we need becomes available over the counter, so we can be our own doctor; we ask friends and family for advice on things too personal or trivial to ask anyone else; we even look for cheaper ways to refill our own printer ink.

Specialists matter; and yet often we are terribly happy to find ways around hiring one.

When it comes to computers, most people are happy to leave all their work to tools written entirely by large companies. Then they stumble through the haystack of features they dont need, to get to the relatively few that are vital to the task at hand.

All in all, there's nothing wrong with this. But over time, options are replaced with requirements, and tools with ecosystems and lock-in, and you are pushed

along a path you really don't want to go. This is the road of relative helplessness, where even learning about the tools you need every day includes nothing about how they work; only how to make them do something until they're redesigned once again to work slightly differently.

Sure, you can pay to have custom software made, but it costs a fortune you probably don't have. And you could use software that is designed more for people who are ready to take charge of their most vital tasks, but it isn't heavily advertised (and you have more questions than answers when it comes to those things.)

Well, where do you get the answers? If you take the usual computer course, these things may be buried somewhere in the footnotes of a string of classes a year or more long. You may learn (and spend time learning) so much more than you set out to-- or have time for.

It's true, there are no shortcuts to necessity; but you can cut through some of the things you don't actually need. This cuts both ways, because you still have to learn the things you actually need; and too often those things are simply not being taught.

Today you will hear that kids need to learn to code. If theyre not going to do it for a living,

exactly what is the benefit?

I will go so far as to say that learning to code is the shortest route to computer literacy. This is for two reasons: first, it supplies a solid foundation. Also, it gives you valuable insights into processes that make up essentially 100% of what happens when you are using a computer.

The thing about coding is you have to experience it to really understand it. A pilot moves a handle in front of them and twiddles controls, while a painter smears liquid onto cloth all day; but to learn to do these things so they produce the desired results requires either an unusual level of instinct and trial and error, or an excellent education; sometimes it takes both.

Not everyone has the instinct or the patience to teach themselves, and many more wouldn't know where to begin. But I have spent the past three decades teaching myself computing and how to write code, and like so many of us at the time, one of the first things I learned was the BASIC programming language.

I spent years looking for a suitable replacement for Basic, weighing things like how easy it was (preferably not even required) to edit existing code to work on newer dialects, how easy it was to write new programs, how easy the language is to

explain, and how fast and fun the process is.

I finally settled on Python, a language whose author now works for Google and whose name is inspired by the british comedy troup.

The reason I was looking for a replacement for Basic, is that Basic development tends to go in one or more of two general directions:

Either it stays true to its design but lacks relevance in its ability or application; and/or it does a great balancing act between tradition and modernity, then finally succumbs to the authors' desire to build the next "easy" game engine, abandoning its original userbase or dragging them through too many irrelevant changes.

Or, you can use Python; which is a lovely language to teach yourself, but installing it (properly) in Windows is a pain (it comes pre-installed on Mac OS/X and most GNU/Linux platforms) and some people get turned off by its case-sensitivity or rules about indenting.

But before we go any further into that topic, let's see what it was like to code in Basic in 1988:

```
10 CLS
20 COLOR 5
30 PRINT "Hello, world!"
```


Notice the line numbers on the side? A relic of the Dartmouth Time Sharing System (DTSS) that ran the original version of BASIC in the 1960s. Today you could run the same program like this:

```
cls
color 5
print "Hello, world!"
```

ALL CAPS is the convention in a number of mid-to-late 20th century programming dialects, including COBOL and anything running on an Apple][without lower-case characters available. Fortunately today, all-lower is something you're more likely to see than all-upper.

Let's look at the three lines of that Basic program source:

```
cls
```

This command means "Clear [the] screen." It blanks the screen and moves the cursor back to the upper-lefthand corner.

```
color 5
```

This one changes the text color to attribute 5, which in this case means that any text that is output next will be in the color magenta.

The colors available by default in DOS (the usual home of Basic in the 1980s) can be found in the following table:

0 Black	**8** Grey
1 Blue	**9** L. Blue
2 Green	**10** L. Green
3 Cyan	**11** L. Cyan
4 Red	**12** L. Red
5 Magenta	**13** Pink
6 Brown	**14** Yellow
7 White	**15** B. White

Higher graphics modes allowed a larger number of colors, but 16 was a good number sometimes.

The third line of that three-line program is:

```
print "Hello, world!"
```

This line makes the program one of the most famous types of program there is: the "Hello, world!" program.

A program that puts "Hello, world!" on the screen is often a first (however useless) example of how a programming language works. Sometimes you can tell how complicated a language is going to be to learn based on this program. (Often you can't tell from a single line like that.)

In summary, the Basic program we listed blanks the screen, changes the text color, and puts the words "Hello, world!" on the screen.

Today, you would probably just create a website or other document containing these words; programming however, allows you to do so much more than write things down.

Incidentally, the reason the "put things on the screen" command is named **print** is related to the standard output technology of the 1960s and 70s: the teletype.

Like a giant typewriter, you would type things onto paper (the computer would know which keys you pressed) and the computer would print its responses afterwards, underneath your input. To this day, text too long to fit on the screen often "scrolls" to simulate the paper on a teletype machine.

Python 2.x and Basic have the same print command:

```
print "Hello, world!" 'Basic

print "Hello, world!" #Python
```

The top line concludes with a comment ' which is preceded by an apostrophe, to let the computer know it can ignore the rest of the text on the line. In Python and a few others, a hash **#** does the same.

In Python 3, the command is **print("Hello, world!") # because parentheses for the print statement are now required.**

Basic went online in 1964 at Dartmouth College, and was designed specifically to make computer programming accessible to everyone. From its beginning as a way for non-computer specialists to make use of computers, Basic spread to secondary schools and even elementary school use.

Another language that was developed in the 1960s is Logo, best known for its triangular graphics cursor and Spirograph-like designs that allow children and adults to learn programming by exploring simple commands and tasks. Unlike Basic, Logo lends itself foremost to graphics while more "practical" applications are relatively difficult to teach.

However, in some ways Logo has survived Basic in educational use; being spun off into "Drag-and-drop" programming environments such as Scratch, TurtleArt and Minecraft.

While these examples may not all be direct descendants of Logo, there is a clear evolution from the commands and exploratory nature of Logo to easily discoverable and graphical programming for complete beginners. TurtleArt in particular follows Logo in design, purpose and output, while Scratch can be seen as an obvious continuation and extension of tasks made trivial by Logo.

One of the problems I find with these environments is that they are so abstract-- they do teach the concepts of programming, but without making the user very aware that they are producing code; It is difficult to instruct an adult in Scratch without their reasonable assumption that they are only automating a cartoon character.

In fact, there is practically nothing you can do with a modern high-level scripting language that cannot also be done with Scratch. But this is not what Scratch is designed to make easy. My interest is in making code itself accessible, not only teaching core concepts through a friendly interface.

From the example of Basic, we know that children can learn to write code from an early age in a school setting. The concept of programming in terms of a language and not just automated tools is still valuable; as is the Scratch platform. But since Basic is (in this author's opinion) overly complicated versus what it was in the 60s and 80s, I set out to create a language with some of the strengths of Basic, Logo and even Python alike.

Originally called "fig basic," the name is now shortened to "fig" and is implemented in Python. While one of the primary goals is to teach basic programming concepts, fig is also designed to acclimate the user or student to a text-based coding environment. Instead of avoiding the

environment that is arguably ideal for hard copies of program source, for logging into a remote machine including the single-board computers like the Raspberry Pi or Beaglebone and Arduino boards, fig caters to that environment and encourages its use.

But since one of the primary goals is to help the user become familiar with core programming concepts as quickly and simply as possible, this will be a focus in the pages to follow. Specifically, fig is based on the following 7 (and fairly universal) concepts:

* Variables
* Input
* Output
* Basic math
* Loops
* Conditionals
* Functions (subroutines)

The goal of coding in fig can be simply to be able to write code in fig; however, if you can code in fig, it can be a tool for learning more advanced languages like Python, or most languages including JavaScript, Bash, PHP, or even C.

More than that, I believe that even proficiency with Basic will lead to a more instinctive

understanding of computing (let alone programming) in general. Fig is one attempt to make something more "Basic" than Basic; which I encourage all coders to try.

Different languages have different strengths, and weaknesses. Most languages you will find in use have enough functionality to simulate most other languages and even simulate computer architecture itself; this is known as being "Turing complete."

"Complete" in this case does not mean trivial; you can use Scratch (by far one of the friendliest programming environments ever devised) or Minecraft to run text-coded programs written in other languages; but getting these environments to behave that way would be anything but easy. Often the "easy" way to achieve a more complicated task is with more complicated software, or a more sophisticated language.

You still have to start somewhere, and "where" is not only the subject of a great deal of debate, but will remain fertile ground for innovation for many years to come.

Chapter 1: Variables

To fully appreciate the power and feel of coding, it is necessary to write or at least "run" code. You can get a glimpse of this from watching someone else run code, but you will not experience the confidence or joy of being able to tell the computer something and have it do what you say unless you actually do it.

One of the goals of fig is to avoid "formulas" by default. In fact you can create all kinds of formulas using fig, and if it feels too limiting you can add Python code inline to a fig program. Python allows full formulaic expressions, of the kind fig deliberately avoids.

By allowing this kind of extension, fig is both friendly and powerful. In fact you could simply start with Python if you had the interest in doing so, but fig will provide a number of friendly shortcuts that allow you to quickly progress without getting slowed down by too many implementation details.

Variables are one of the most universal aspects of programming, and unlike the Basic example in the previous section, fig requires you to either use Python or a variable to create a "Hello, world" program.

A few keywords in fig are designed to stand alone, but in general most lines of fig code begins with a name supplied by the user. You can use a letter, like "z" or the word "Now" (fig is not case-sensitive, so "Now" and "now" and "NOW" refer to the same variable) but this "main variable" is a feature of fig that often saves you the trouble of referring to it several times in the same line.

To return to our Basic example:

```
print "Hello, world!"
```

Fig is designed to be read left-to-right, and does not use parentheses (at least they do not have syntactic importance; they are decorative, and strictly for the user to determine the significance of.) In fig, the above line would start with a variable. For the moment, let's just use "Now":

```
now "Hello, world!"
```

In Basic, this would be **now$ = "Hello, world!"** and in Python it would be the same, except for the **$** after the word "now."

In fig this creates (or resets) a variable named **now** – which also gets set to a numeric value of zero (0.)

The next part of the line changes the value from 0 to a "string" of characters in quotes, in this case the phrase **"Hello, world!"** So far, nothing visible is done with this information. The program sets the value of a single variable, and does nothing else. But it goes left-to-right very clearly:

* The first bit of text names and creates a variable.
* The second bit of text changes the information (the value) that is stored by the variable.
* The next thing we add to the line will probably do something with that variable, too.

```
now "Hello, world!"  print
```

Okay, so now we've added the **print** command, same as Basic and Python. And so our variable **now** is output ("**print**ed") to the screen.

You can do other things with the **now** variable; you could make it all upper-case, before or after it **print**s, using the **ucase** command:

```
now "Hello, world!"  print  ucase
```

This changes the value to upper case, but (left to right) it prints before that happens, so no change is yet visible. This:


```
now "Hello, world!"  ucase  print
```

...changes the value before using **print** and so you can see the change in the output.

Fig is different than Basic in that the most obvious or typical example (Hello, world) is made purely of output statements in many languages. In fig, you begin with variables right away.

This is primarily done for friendliness, but it also means that your first program will likely contain 2 of the 7 core concepts, (variables and output) instead of just output. Although this is functionally similar or identical to a formula, I think it's a better introduction to what programming is really about.

Fig doesnt actually avoid the functionality of a formula, but the parenthetical structure that makes it look "too much like math" to the math-phobic. It is still mathematical in nature, but closer to natural or spoken language in syntax and structure.

Copying variables is much like setting them.

If you want to copy the value of the variable **now** into say, **another**:

```
now "Hello, world!" ucase
another now
```

So **now** is still **"Hello, world!"** but so is **another**, and you can use/change each variable independantly (or use one as a backup, to check later if the other has changed.)

Each of the two lines above start with a "main variable" and the program code on each line works with that variable:

* Putting a value after the main variable (this is optional) will set it

* Putting another variable (also optional) after the main variable will copy its value into the main variable.

* Some commands will read/use, but not change the main variable; **print** is an example.

* Some commands will write/change the main variable, but not read it.

* Some commands will read and change the main variable-- **ucase** is such an example.

* Some commands will not read or change the main variable, such as **cls**; it does the same thing that **cls** in Basic does.

Most commands in fig share a line / begin a line with a main variable, which is not optional. When fig gains a new feature, it is generally a "shared-line" command, or not a program keyword at all.

However, a few keywords do not share a line; they get their own line and do not require (or use) a main variable; commands/keywords of this type are not generally added to fig.

These "own-line" commands are:

graphics / textmode

for / forin

while
break
pass

function
python
fig / next / nextin / wend

iftrue / ifequal / ifmore / ifless / else

try / except / resume

These commands are so important, they get their own line-- if only to stand out as part of a command block (a multi-line series of commands) or as a

command that significantly changes the behavior or mode of the program.

All other commands require a line that starts with a main variable:

now cls # cls is a command, now is main var

In the above example, **#** is a special case-- it tells fig to ignore the rest of the line so that it can be used to "comment" the program. Technically **#** is a command or keyword, but it can be on a shared-line with a main variable, or begin a line, or be on a line by itself.

However, it also isnt really part of the program; it just begins a note inside the program code.

If you put a **#** in front of a line of working code, it will turn that line into a comment and thus disable the line or stop it from running as part of the program. This can be useful for testing the functionality of a single line or group of lines, and using it this way is called "commenting out" a line.

What else can be said about variables? So far, we have only demonstrated keywords that work without any additional details in the way they are written: most keywords, like **cls** and **print** and

ucase are simply tacked on to the rest of a line, at the right of the line. The program goes from left to right, and then to the next line.

Some keywords go on their own line, such as **textmode** or **while**, and do not share a line with a "main variable" to start, or with other commands. But we haven't talked about parameters yet.

Apart from the start of a line, or the second "word" in a line (where it is used to copy one variable's value into another) the place you are most likely to see a variable is as the parameter for a command that has them.

Most fig commands have zero parameters, or one parameter. A few have more than one, but each command has the same (fixed) number of parameters in every context, except for **function** (used to define new fig commands.)

Therefore **cls** always has zero parameters; **while** always has zero parameters; **ucase** always has zero parameters, and **textmode** always has zero parameters. The keywords **left** and **right** always have a single parameter; it defines how many characters you want to keep from one side of a string.

To demonstrate:

```
now "hello"    print    left 2    print
```

First **now** is created, set to **"hello"**, then **print**ed, then **left 2** is run before the result is **print**ed again. The output is:

```
hello
he
```

We've used four spaces to separate each command, though one space is sufficient.

The **left** command includes **2** as a parameter, as the number of characters to take from the side of the string that's stored in the main variable. It then changes the main variable to the leftmost 2 characters.

Since **print** is called before and after **left 2** runs, it displays the value of **now** before and after as well.

Like most commands that have parameters, **left** does not have to use a "constant" value like **2**, but can use a variable instead:

```
howmany 2
now "hello"    print    left howmany    print
```


Using variables as parameters allows the program to have more control at run-time (it allows it to be more flexible and powerful) and is somewhat the essence of things being "programmable." It allows the program to take input and have it affect the output, even before we get into conditionals.

As mentioned, the extra spaces are optional command separators (purely for the visual aspect of the code) and each of these lines are valid and do the same thing:

```
now "hello"    print    left howmany    print
now "hello" print left howmany print
now "hello" : print : left howmany : print
now "hello" ; print ; left howmany ; print
```

The colon **:** is the traditional command separator in Basic, and in C and Python and JavaScript, a semicolon **;** is used.

It is worth talking about variable types (or rather data types, since any variable in fig could hold one or more type of value) but the subject is simple enough that the point of mentioning it might not be obvious:

* **string type:** data "in quotes" is a string of characters, which may include numbers but they will not be treated as a numeric value. **"5" plus "5"** returns **"55"**

* **integer type:** an integer, also known as a "whole number," is a numeric and has no decimal point. **5 plus 5** returns **10**

* **float type:** numeric and includes a decimal point; allows decimal values. Like Basic, fig handles floats and integers pretty seamlessly, but strings have to be converted using the **val** command to be treated as numerics. **5 plus 5.7** returns **10.7**

* **array type:** while variables hold one value at a time, a variable can be converted to an array holding the same value, plus others.

```
now 5  arr  # create an array holding 5
now 5  arr  times 100  # array holds 100 5s
```

Normally the **times** command is used for basic multiplication, but when used on a string it sticks several copies of that string together into one:

```
p "hello" times 4 # "hellohellohellohello"
```

When you start a line with a main variable that you've already used, it sets it to zero:

```
p  print          # displays 0
p  5  print       # displays 5
p  print          # displays 0 again
p  plus 7  print  # displays 7
```


Arrays are the exception to this rule. Since they might hold a lot of information, they do not get cleared on use as a main variable unless you explicitly clear them:

```
p 5  arr print    # displays [5]
p  print          # displays [5]
p 0  print        # displays 0
```

5 plus "5" plus 5.0 returns **[5, '5', 5.0]**

Arrays in fig are the same as arrays in Python, known in Python as "lists." Lists in Python (like in fig) are not restricted to a single type per array; they can hold a mix.

Apart from converting a variable to array using arr, fig has several ways of creating an array:

* the **split** command can split a string into array elements.

* **arropen** and **arrcurl** load the lines of a file or webpage into an array, respectively.

* **arrstdin** creates an array from information "piped" in from another program. This allows you to mix the functionality of various programs together, even if they aren't designed as a group or suite.

* **command** creates an array of parameters the fig program was called with. These are not parameters of individual commands, but of the program itself. (This allows you to put together your own "language" from fig programs and others, such as bash utilities)

* inline **python** can be used to create arrays

* **arrshell** runs command line programs, and returns the output of those programs as an array.

Chapter 2: Input

Some functionality is a matter of both input and output; in fig, most or all functions can be separated into or at least categorized as one or the other.

For example, the previous chapter mentioned **arrshell**, which is categorized as an input command because it takes all the output of a command line shell and puts it into a variable. Even if the shell session does other things, the purpose of **arrshell** is to collect and store information in the program itself.

However **open** is categorized simply as a function, because whether it opens a file for input or output depends on the value of its single parameter. It is not an input command when it opens as output, and it is not an output command when it opens as input, but it is always a function so it is categorized that way.

You can use the **function** command to define routines or commands that perform input, or output, or both or neither. The **function** command is very obviously categorized as a function (related to functions.)

Now that we've meandered over that distinction, let's talk about fig commands that are clearly related to input.

```
x  timer
```

sets **x** to the number of seconds past midnight. It gets the value using the computer's internal clock.

```
x  arrstdin
```

sets **x** to an array of all information streamed into the program from whatever program called it. This can be used to "chain programs together" including programs written in other languages, by other people. In Unix-like operating systems and even DOS and Windows, this is called "piping" information from one program to another.

To pipe information from another program to a fig program, use **arrstdin** to get the information into an array.

To pipe information from a fig program to another program, simply use **print** on the fig side, and whatever functionality accesses stdin on the other end. On the command line:

```
 #optional     -->                    --> #optional
non-fig-program | ./fig-program.fig.py | other
```

The **./** means "run this from the current folder" and is meant for calling programs in GNU/Linux, BSD, and OS/X.


```
x  lineinput
```

sets **x** to the string that is typed in on the keyboard. The program waits for Enter to be pressed.

```
x  "text.txt"  open "r"
y  flineinput  x
```

opens "text.txt" for input and reads a line using **flineinput** which gets stored in the variable **y**. If you had an incredibly large file and you didn't want to open all of it at once, this would be one way to do that. Otherwise, **arropen** is simpler:

```
x  arropen "text.txt"
```

opens **"text.txt"** into array **x**. really, that's all you do. If you used open "r", **run "text.txt" clo**se

```
x  time
x  date
```

While **timer** gets the seconds past midnight, **time** gets the HH:MM:SS time and **date** gets the MM/DD/YYYY date. Unlike **timer** which is numeric, both **time** and **date** return strings.

x arrcurl "https://duckduckgo.com"

downloads the source of the page for https://duckduckgo.com and loads it into an array just like **arropen** would for a local file.

x sleep 2

waits until **2** seconds have gone by to continue running the program. Does not affect the main variable, and is only categorized as "input" because it gets information from the clock.

x command

sets **x** to an array including each parameter the fig program was called with. Didn't open the program you wrote with any parameters? Then this command isn't going to do much for you.

If you are on the command line, this is how you call your program with parameters:

name-of-your-program.fig.py par1 par2 par3

The array will contain "par1", "par2" and "par3".

Chapter 3: Output

Where input commands get information from a device (or file, or connection, or from some part of the computer,) output commands send information to a device, or file, or connection, or to some part of the computer.

Obvious output devices are the screen and speakers. Sending a file to the printer counts as output, although the printer itself also sends information back to the computer, making it an input/output device.

A touchscreen is a separate device than the screen itself and an input device, despite the fact that the display directly behind it is for output. These are technical distinctions and not always important, but they are relevant to programming.

x "hello there" print

sets **x** to a string and sends **x** to stdout, which generally means the screen (or window.)

x "hello there" prints

same as the previous line using **print**, except **prints** stays on the same line instead of advancing.

```
x    "text.txt"  open "w"
y    "ok"        fprint x
now  x           close
```

You would normally see these used separately.

The first line sets x to a string containing the path and filename of the file to save information to; then it opens the file for writing (the "w".)

The **fprint** command writes the value of the main variable to **x**, which is still holding the filename. In this case, the string "ok" is saved.

The **close** command will update / save / stop writing to a file that was opened before. It closes whatever file is specified by the main variable. Don't **close** the file until you're done **fprint**ing (each and every line you want to save) to it.

Most languages use a filenumber or handle to track an open file; which is a different thing than a string or a numeric variable. I've always thought that is weird or unfriendly, but the solution in fig seems fairly unique (and unfamiliar if you're used to Basic or Python.)

Fig keeps the (necessary) handles in a special type of array called a dictionary, with strings as the key to each handle. You don't worry about any of

this, you just make certain to use the same string to **close** the file that you used to **fprint** (or **flineinput**) and **open** the file in the first place.

In other words, if you use the string "../text.htm" with the **open** command, you will need exactly the string "../text.htm" to read or write or **close** that file. It doesn't matter if it's a constant (written out) or two variables holding the same value, as long as it's a perfect match.

For what it's worth, ending the program with **end** or **system** (or just letting the program stop normally) is supposed to close all open files.

x cls

clears (and goes to the top left of) the screen.

x display

first time called, turns off auto-update of graphics (default is auto-update.)

If already called once, updates graphics (only affects real graphics mode, not textmode graphics.)

textmode

forces text-mode graphics; don't try to display graphics commands in a graphics window; display them using ansi escapes.

graphics

default mode; open a graphics window for running graphics commands. If graphics window fails (perhaps optional pygame is not installed) then falls back to **textmode**.

If **textmode** was used explicitly, **graphics** command turns real graphics back on.

x line 5 7 10 20 14

draws a line from (5, 7) to (10, 20) in yellow.

x pset 5 7 10

draws a point at (5, 7) in light green.

x locate row column

move to (row, column) on text screen.

x colortext 5

changes text color to magenta:

0	Black	**8**	Grey
1	Blue	**9**	L. Blue
2	Green	**10**	L. Green
3	Cyan	**11**	L. Cyan
4	Red	**12**	L. Red
5	Magenta	**13**	Pink
6	Brown	**14**	Yellow
7	White	**15**	B. White

x highlight 0

changes text background to black:

0	Black	**4**	Red
1	Blue	**5**	Magenta
2	Green	**6**	Brown
3	Cyan	**7**	White

Chapter 4: Basic Math

```
y 5  plus 12  times 3
```

adds 12 to 5, then multiplies times 3.

Fig always goes left to right. If you want to do parentheses and order of operations, use Python:

```
y 0  # most important part right here
python
    y = (5 + 12) * 3  # 51; different than:
    y = 5  + 12  * 3  # 41; (order of ops)
fig
```

set **y** in Python, to 5 + 12... then * 3

Then reset **y** as 5... + (12 * 3) or: 5 + 36

Inline Python, unlike the rest of the fig language, requires accuracy in the indentation. Lines begin at 4 spaces from the left, and each indent increases by 4 spaces.

Python is not actually part of the fig language; but fig allows you to include snippets of Python in your fig programs. Since fig translates to Python, those sections will be included un-translated.

Notice that before **y** is used in the Python code, fig sets **y** to **0** before switching to inline Python.

```
y 0
```

ensures that fig knows **y** is a valid variable in use, so that after Python uses it, fig already has it registered and will use the value Python left it with (instead of thinking **y** is unused.)

You can avoid this extra step if you know that **y** is already:

* used as a main variable in fig at least once
* has a value that the Python code can use or reset

```
y 0
python
    y = 5
fig
x y  minus 2.5  print
```

sets **y** to **0**, uses Python to change it to **5**, sets **x** to **y minus 2.5**, (which is 2.5) and **print**s **x**.

plus and **times** also work on strings and arrays.

```
x  minus 5
```

sets **x** to -5.

```
y 200
x y  divby .5
```

sets **y** to **200**, copies **y** to **x** and divides by **.5**

```
x 25  oct print
```

Why do programmers mix up Halloween and Christmas?

Because oct 31 is dec 25.

Decimal		**Oct**al		Hex	
0	15	0	17	0 f	
1	16	1	20	1 10	
2	17	2	21	2 11	
3	18	3	22	3 12	
4	19	4	23	4 13	
5	20	5	24	5 14	
6	21	6	25	6 15	
7	22	7	26	7 16	
8	23	10	27	8 17	
9	24	11	30	9 18	
10	**25** <--	12	**31 <--**	a **19**	<--
11	26	13	32	b 1a	
12	27	14	33	c 1b	
13	28	15	34	d 1c	
14	29	16	35	e 1d	


```
x 255  hex
```

sets **x** to "0xff"

```
x 50
y 70
r 40
x2 3.14  cos  times r  plus x  int
y2 3.14  sin  times r  plus y  int
```

sets **x2** to the cosine of **3.14** radians, multiplies that by **r**, and adds **x** then converts to an integer.

Then sets **y2** to the sine of **3.14** radians, multiplies that by **r**, and adds **y** then converts to an integer.

If instead of **3.14**, you use another variable that loops from **-3.14** to **3.14**, **this will plot a circle** with the center (**x**, **y**) on points (**x2**, **y2**).

```
x 1  atn  times 4
```

sets **x** to the arctangent of **1** and multiplies that by **4**, which gives Pi to 15 decimal places in Python 3. (11 in Python 2, in both the same as **math.pi**).

```
x 3.14  tan
```

sets **x** to the tangent of **3.14** radians.

x 2.5 int

sets x to 2.5 and converts to an integer.

x -5 sgn

changes x to either -1 (if the value is below 0) or 1 (if the value is above 0.) if the value is 0, it stays the same.

In this case, x becomes -1.

x 25 sqr

sets x to 25, and then to the square root (5).

x 255 mod 7

sets x to 255, then to 255 modulus 7.

x 1024 topwr 2

sets x to 1024, then 1024 to the power of 2.

Chapter 5: Loops

This is really the first chapter about "program blocks."

So far, we've only covered one type of block: a snippet of inline Python:

```
python
    y = (5 + 12) * 3  # 51; different than:
    y = 5  + 12  * 3  # 41; (order of ops)
fig
```

This block begins with the **python** command, and the end is marked with the **fig** command.

The defining characteristic of a block is probably that **it starts and ends with a pair** of commands.

In fig, you can end any block with the **fig** command. But semantically it makes the most obvious sense when it ends a **python** block, because **python** means: "here is python code" and **fig** means: "get back to **fig** code."

All loops are program blocks, also known as "command blocks."

Between the start and end of the loop blocks are the lines of code that will "loop."

Fig has three kinds of loop: **for**, **while** and **forin**.

The simplest loop is the **while** loop. Let's make a single line of fig code to print a zero:

```
x  print
```

now let's start a **while** block by putting it on the line before the one we just wrote:

```
while
x  print
```

We need to mark the end of the loop, or fig will loop **every line** that comes after the **while** command:

```
while
x  print
fig
```

There is nothing wrong with this block; it's perfectly formed. However, **while** blocks have their own end marker, which you can optionally use in place of the standard **fig** command. It does exactly the same thing, but helps note which block it ends:

```
while
x  print
wend
```


So you can end a **while** block using **fig**, or you can end it using **wend** (it stands for while-end.) The choice is up to you, they are interchangeable.

The **while** command exists in Basic, Python and even C. In Basic and Python, you can use **while True:** or **while 1:** to keep looping until the loop breaks with **break** or (in Basic) **exit while**.

Since for more than half a decade, I've used **while 1** in Python instead of worrying about setting up the loop with a condition, fig is designed this way for simplicity.

```
while
x  print
wends
```

prints 0 (or an array stored in **x**, where applicable) repeatedly until the user breaks with ctrl-c on the keyboard.

The other way to break out of a **while** loop is using the **break** command (same as in Python) but without a conditional (those are covered in the next chapter) a **break** command will either prevent the loop itself, or even prevent the lines inside it from running:

```
while
x  "this only runs once-- no loop"  print
break
wend
```

The **break** command stops the loop right after it runs the code inside it once, so it is as if the loop isn't there at all; only the line that prints.

```
while
break
x  "this line doesn't run at all"  print
wend
```

Moving the **break** command to before the other line inside the loop exits before the lines after it can even run-- so this entire block does nothing other than waste a tiny amount of time.

In order to make a **while** loop do something useful, you probably want to put the **break** command inside a conditional block. We'll get to those in the next chapter.

A **forin** loop will loop through the items in an array, doing the same code in each one. These items can be lines of a file, lines in a website, letters of a string, pieces of a string made into an array by split-- any array.

Let's do words in a string, by splitting the string into an array by spaces:

```
names "Ady Susan Kerry Morgan Lori" split names " "
forin p names
    now p print
    fig
```

fig will **split** the string by **" "** to make an array called **names**, and **print** each one:

```
Ady
Susan
Kerry
Morgan
Lori
```

We end the **forin** loop using **fig** again, but just as **while** has **wend** for (optional) semantic use, **forin** has **nextin** (or **next**). Use whichever you prefer.

```
forin p names
    now p print
    nextin
```

If you want to loop through numbers, you can use a **for** loop. A forin loop will run the same code repeatedly, once per array item, and a **for** loop once per number in a range-- **for** has 4 parameters:

* the variable for will set with the value of the current item (just like a forin loop.)

* the number to start with

* the number to stop at

* the "step" or number to increase on each loop

In other words:

```
for v start stop step
```

Suppose we want to bring variable size from 5 to 12, doing every whole number in that range:

```
for size  5 12  1
    now size  prints " " prints
    next
```

will output: 5 6 7 8 9 10 11 12

Odd numbers from 33 to 17?

```
for s  33 17  -2
    now s  prints " " prints
    next
```

33 31 29 27 25 23 21 19 17

A funny quirk of Python is that it insists on integer (whole number) steps in numeric ranges. In Python (which is what fig translates to,) a for (range) loop isn't going to do a decimal step.

```
for v start stop step # step has to be integer
    now v print
    next
```

However, if you use a constant for the step instead of a variable, **fig** will let you do a float step:

```
start 5
stop 10
for v start stop 2.5 # step is constant 2.5
    now v prints " " prints
    next
```

outputs: 5.0 7.5 10.0

This builds a different kind of loop (actually a custom **while** loop) in the Python translation, instead of a **for** loop.

In the next chapter, **while** loops will become more useful when they can start and stop based on conditions. Certainly you can use them anyway, in any situation where using ctrl-c on the keyboard to stop looping is suitable.

Chapter 6: Conditionals

We've covered a couple kinds of block so far:

* **python** marks lines written in Python, then **fig** ends the block

* **while** marks lines that will loop, then **wend** marks the bottom of the lines that will repeat

* **forin** and **for** mark the beginning of a loop through numbers or arrays, and **nextin** or **next** marks the bottom of those loops

now we have another kind of block:

* a condition describes some aspect of the state of the program, and the lines inside the block only run if the condition is satisfied / true

like with **python** these blocks will end with the **fig** command.

Here is a very simple example:

```
iftrue 1
    now "that is true"  print
    fig
```


An **iftrue** conditional has one parameter, and if the parameter is "true" (non-zero) then it will run the code inside the block. (between the conditional and the **fig** command.)

In the previous example, 1 is non-zero so it runs the code in the block:

```
now "that is true"  print
```

What isn't true? A zero, or a zero-length string:

```
iftrue ""
    now "this will not print"  print
    fig
```

So, what about iffalse? For that, fig has a command that will invert a conditional, but first let's talk about blocks without code inside:

```
iftrue ""
    fig
```

will give you an error message when it runs. Use **pass** as your placeholder for actual code:

```
iftrue ""
    pass
    fig
```

The ifequal conditional will compare two parameters, and is true if they are equal:

```
x "When is your birthday?" print lineinput
d date

ifequal x d
    pass
else
    x "A very happy Un-Birthday to you!" print
    fig
```

This program (assuming you type the date in using MM/DD/YYYY format) asks you to type in your birthday, and if the date is not today's date, it wishes you a very happy Un-Birthday.

The **else** command separates a conditional block into a two-part conditional block:

* if the first part is true, it runs the code between the condition, and **else**
* if the first part is NOT true, it runs the code between **else** and the bottom of the conditional.

Let's use the **randint** function with its "lowest" and "highest" parameters to produce a number between 1 and 10. Then ask the user to guess the number:


```
x  randint 1 10
y "Guess a number from 1 to 10: " print lineinput

ifequal x y
    now "You guessed correctly!" print
else
    now "Good try, but you didn't guess it." print
    fig
```

Something is wrong with this program, and it will never work the way it's intended to. But we can fix it!

The randint command returns an integer, but lineinput returns a string. Meanwhile, when ifequal compares an integer and a string, it will NOT detect a match even when one is string "5" and the other is numeric 5!

We can fix this by making the value from randint a string using **str**, or making the output from **lineinput** into a numeric using **val**. Since we don't know what to expect from **lineinput** (the user could type anything) we will convert the **randint**:

```
x  randint 1 10  str
y "Guess a number from 1 to 10: " print lineinput
ifequal x y
    now "You guessed correctly!" print
else
    now "Good try, but you didn't guess it." print
    fig
```

By adding **str** to the end of the **randint** output, we ensure that when **ifequal** compares it to the number the user types in, at least a string is being compared to a string.

There are other improvements worth making, but for now this is good enough. (What commands will compensate for extra spaces on either end of the input?)

This is a great time to add a **while** loop. Currently the user only has one guess, and we could give an exact number of guesses with a **for** loop, but that wouldn't demonstrate a **while** loop doing something truly functional, would it? Let's give unlimited guesses, so you can see how to do that:

```
while
    x  randint 1 10  str
    y "Guess from 1 to 10: " print lineinput

    ifequal x y
        now "You guessed correctly!" print
        break
    else
        now "Good try, though." print
        fig
    wend
```

4 things to notice here: The **while** at the top-- the **wend** at the bottom-- plus, a correct guess stops

the loop using **break** after telling you that you guessed correctly.

But perhaps the most important thing is the line with **randint** on it. Because once again, it will not work (exactly) the way that we intend it to:

```
while
    x  randint 1 10  str
    y "Guess from 1 to 10: " print lineinput
```

Do you see what's happening? It gets a random number and gets the user to input a guess. Ok... When it loops, it gets another random number!

Since the number is always 1-10, the user can just keep guessing 5 (or any other number in range) until the computer picks **the user's number** randomly! So let's have it pick the number ONCE:

```
    x  randint 1 10  str
while
    y "Guess from 1 to 10: " print lineinput

    ifequal x y
        now "You guessed correctly!" print
        break
    else
        now "Good try, though." print
        fig
    wend
```

Now it works the way it is designed to: the user has to keep guessing a different number until they can guess the one the computer picked.

But now that you've seen the while loop work in a way that the program can get out with a conditional break, let's replace it with a **for** loop so we can limit the number of guesses the user gets:

```
guesses 4
    x  randint 1 10  str
for r  1 guesses  1
    y "Guess from 1 to 10: " print lineinput

    ifequal x y
        now "You guessed correctly!" print
        break
    else
        now "Good try, though." print
        fig
    next
```

We didn't have to change **wend** to **next**, it just looks more reasonable that way. Also we didn't have to make a variable called **guesses**, we could've just put a 4 there. But this is better (and clearer.)

Let's take a look at the output this program will show the user, along with the numbers the user might type in:


```
Guess from 1 to 10:
5
Good try, though.
Guess from 1 to 10:
2
Good try, though.
Guess from 1 to 10:
8
You guessed correctly!
```

How about another condition to say the user ran out of guesses? The **for** loop keeps track of which guess the user is on, in the **r** variable, right? We can use that:

```
guesses 4
    x  randint 1 10  str
for r  1 guesses  1
    gsss guesses  minus r  plus 1  str
    y "You have "  plus gsss plus " guesses." print
    y "Guess from 1 to 10: " print lineinput

    ifequal x y
        now "You guessed correctly!" print
        break
    else
        now "Good try, though." print
        ifequal guesses r
            now "Sorry, no more guesses." print
            fig
        fig
    next
```

Here are the relevant lines of the changes we made:

```
guesses 4
    x  randint 1 10  str
for r  1 guesses  1
    gsss guesses  minus r  plus 1  str
    y "You have "  plus gsss plus " guesses." print
```

Also:

```
        now "Good try, though." print
        ifequal guesses r
            now "Sorry, no more guesses." print
            fig
```

These two changes don't need each other; the second change would still work by itself, and the stuff at the top could work without the bottom part.

Let's go over what we know:

* the **for** loop will repeat (**guesses**) times, which in this case is **4**.

* the loop tracks which repeat (guess) it's on, in the variable **r**.

* so if **r** is the same number as **guesses**, it is on the last guess (in this case, it was already used.)

That pretty much explains the part at the bottom. It doesn't run until after the program tells the user they guessed wrong, so if **guesses** is the same number as **r**, the last guess was used.

What about the top part? This is just addition and subtraction:

```
    gsss guesses  minus r  plus 1  str
    y "You have "  plus gsss plus " guesses." print
```

* **gsss** is set to the limited number of **guesses** (4.)

* then **r** (which guess user is on: 1 or 2 or 3 or 4) is subtracted, so **if guesses is 4** and **r is 1** (first guess) **then there are 3** guesses left.

* **but...** the user hasn't guessed yet! So we **add 1 more:** "plus 1" since guess 1 is still in play.

* then we convert that number to **str** and put it in a string between "You have " and " guesses."

* then we **print** it.

From the top, it looks really complicated. But if you want this functionality, it's just:

```
gsss guesses  minus r  plus 1  str
y "You have "  plus gsss  plus " guesses."  print
```

You may find with logic like this, that the stricter syntax in Python is easier to work with. Here is what that looks like:

```
guesses = 4
from random import randint
x = str( randint(1, 10) )

for r in range(1, guesses + 1,  1):
    gsss = str(guesses – r + 1)
    y = "You have " + gsss + " guesses." ; print(y)
    y = "Guess from 1 to 10: " ; print(y)
    y = input()

    if x == y:
        print("You guessed correctly!")
        break
    else:
        print("Good try, though.")
        if guesses == r:
            print ("Sorry, no more guesses.")
```

Some of the differences between fig and Python:

* Python cares what case you use (normally all-lower. But Hi and hi are two different things.)

* Parentheses are not optional (in fig they are.)

* You must use = to set a variable, and == to compare (fig requires neither; =can set variables.)

* Functions(are(parenthetical + str(that) + can())) #### become complicated.

* Instead of starting blocks with one command and marking the bottom with **fig** or another command, You have to indent (in fig it is optional) everything after a block-starter. When you are done with the block, you unindent.

* Colons at the end of a block-starter are generally required in Python.

* Python 3 and Python 2 use print differently.

* Commands like randint have to be **import**ed.

Believe it or not, this isn't meant to discourage you from using Python. If you're comfortable with mandatory indentation, it can be very beautiful (I wrote fig in Python, so I know it's a great language.)

But when I tried to teach Python to people, I noted what gripes came up the most. Mandatory indentation is good for some and terrible to others. Fig doesn't have it. Case-sensitivity is fine for most coders-- I prefer the way Basic was not case-sensitive (in Visual Studio I believe it is now.)

If you love Python, on the next page is an approximation of the same code in fig, using optional () and = and : and such:

```
# a program in pure fig code
guesses = 4
x = randint(1, 10) ; str()

for r (1, guesses,  1):
    gsss = guesses ; minus r ; plus 1 ; str()
    y= "You have ";plus gsss;plus " guesses.";print
    y = "Guess from 1 to 10: " ; print()
    y = lineinput()

    ifequal x, y:
        now = "You guessed correctly!" ; print()
        break
    else:
        now = "Good try, though." ; print()
        ifequal guesses, r:
            now = "Sorry, no more guesses." ; print
            fig
        fig
    fig
```

Obviously you can't dress up fig exactly like Python (without putting it between **python** and **fig** that is) but you can dress up fig, if you want to.

Whatever you consider "simple" is what you should stick to code-wise, until you are ready and interested in getting more complicated. When you're ready, there are lots of things to try, including making new fig commands using **function** and/or inline Python. And there are more conditionals!

To review: **else** inverts a conditional, and **pass** is a placeholder (in case you don't want to run code for every condition, but you do want to run code for some.)

Using a comment # instead of pass will not work in loop and conditional blocks.

```
ifequal  x  y
    now print "yes"
    fig


ifmore   x  y
    now print "yes"
    fig


ifless   x  y
    now print "yes"
    fig
```

The **ifmore** (x, y) conditional is true only if x is more than y.

The **ifless** (x, y) conditional is true only if x is less than y.

Substitute x with some other variable or value; and substitute y with some other variable or value.

Once you understand conditionals, else and pass, there is a very useful conditional-like block that is very special: the "condition" it handles is an error.

It works like this:

```
try
    pass
    # try to do this
except
    pass
    # run this code if there was an error before
    resume
```

resume is another subtitute / alias for **fig** which you can use instead (as usual) if you prefer.

```
try
    x  5  divby 0
except
    x  "You can't divide numbers by zero."  print
resume
```

catches a real error.

You can use a try / except / resume block to try opening a file that isn't available (or that the user didn't spell correctly) or put it in a loop

that tries to convert user input to a numeric, and asks again if it can't:

```
while
    x "Please enter a number: "  prints  lineinput
    try
        y x  val
        # if that didn't trip except, break loop
        break
    except
        now ""  print  "That's not a number." print
        resume
    wend
now y  str  plus " times 5 is: "  prints
now y  times 5  print
```

That functionality we just created could be used more than once in a program tha asks for numbers. Let's simplify it before getting into functions:

```
while
    x "Please enter a number: "  prints  lineinput
    try
        y x  val
        break
        except
        now ""  print  "That's not a number." print
        resume
    wend

now y print # this is the number we know is numeric
```

Chapter 7: Functions

In the previous chapter, we put together some code that asks for (and insists upon) a number.

If you've ever been to a website that rejected some input because it wasn't suitable, the code that does that is very similar! (It's probably written in JavaScript or PHP though, because those are still more common languages for websites.)

We have learned several fig commands, and there are several more to learn in this chapter, but for the moment we are going to learn how to **make new ones.** First here is the code from before:

```
while
    x "Please enter a number: "  prints  lineinput
    try
        y x  val
        break
        except
        now ""  print  "That's not a number." print
        resume
    wend
now y print # this is the number we know is numeric
```

See the **now y print** at the bottom? The variable **y** is the one we get when things go right. Keep that in mind.

To create a function (a new fig command) called **thisthing** which uses one parameter: **x** and converts it to the cosine of **x** radians, do this:

```
function  thisthing x
    now  x  cos  return now
    fig
```

Ok, but that's not what we're doing. We are going to create a function called asknumeric with zero parameters.

We define our function using a **function** block:

```
function asknumeric
    # we really need the pass command here.
    fig
```

And the code inside the block will produce a variable called **y**, so let's ask the function to **return** that value (as the function's own value)

```
function asknumeric
    # our code will go here
    now return y
    fig
```

On the next page, let's put the two together:

```
function asknumeric

while
    x "Please enter a number: "  prints  lineinput
    try
        y x  val
        break
        except
        now ""  print  "That's not a number." print
        resume
    wend
now y print # this is the number we know is numeric

    now return y
    fig
```

We can get rid of the "**now y print**" line:

```
function asknumeric
while
    x "Please enter a number: "  prints  lineinput
    try
        y x  val
        break
        except
        now ""  print  "That's not a number." print
        resume
    wend
    now return y
    fig
```


Okay, run the program...

What? It doesn't do anything. Congrats! It's your first custom-made fig command.

You've defined it using **function** but you haven't used it yet!

Your new function is called **asknumeric** and always returns a number, so you can use it like this:

```
x  asknumeric  times 1000  print
```

See? Any time you want to use your **asknumeric** command, just use it like you would any other fig command. (It will only be available in programs that contain the definition before you use it.)

You can even create a user-defined function that calls another user-defined function:

```
function doitagain
    x  asknumeric  times 1000  print
    fig

now doitagain
```

Let's run this and on the next page, see what happens...

Please enter a number: Okay

That's not a number.
Please enter a number: Yes it is!

That's not a number.
Please enter a number: a number

That's not a number.
Please enter a number: 5.78
5780

One of the more interesting things about writing programs is coming up with names for the different variables (and functions.)

If you pick a name that is too short or not very descriptive-- x for example, is really only "descriptive" if you're describing a point on a horizontal axis-- then you cause anyone reading your code to suffer a little.

If you pick a name that_is_too_absurdly_long then you cause anyone re-using (referencing) that name to suffer. People don't seem to mind this as much (I do, so I tend not to use very long names.)

If you're writing a manual, it matters somewhat less because people know "x" is always a variable.

But coming up with names is "work" sometimes.

What's more, is coming up with names can get in the way of more pressing / important matters; so anything that helps reduce the amount of "overhead" in terms of organizing used and unused variable names is an idea worth considering.

The thing about functions is "scope." If you are defining a function, the name you use to call (reference, refer to) the function-- its name-- is the thing you have to put the most thought into.

Everything inside the function is more or less isolated. This isolation is called "scope," and it can be viewed as an inconvenience or an enormous help.

If you're writing a large, complicated program, organizing everything into functions is the best thing you can do. (You can also use objects, which are more complicated, and then "functions" will be called "methods" as far as terminology goes. But fig doesn't really get into objects.)

Because of this "scope" (the way it isolates the stuff inside the function block) you USUALLY need a way to get information from the program into the function.

In our first example, information gets into the function by way of the **lineinput** command. But if you refer to a program variable from outside the function, it will not know anything about it.

Put another way: a variable called "cheese" outside the function block is going to have a totally different value / significance / relevance as a variable **with the same name** inside the function.

And this is so when you're creating a function, you don't have to know what variables are used in the program that calls it.

For all you know, "x" is the most important variable in the whole program. So if you need a variable named "x" in your function, it's completely separate; the two don't interfere. Every function can have its own "x" variable.

And every function indeed does get its own variables. But what if you **want to** get information from the program into the function? This is done with parameters. Lets call them **thing1** and **thing2**. (You can have 0 or more parameters, even 20 of them. However the smaller the number, the better.) Parameters don't need numbers by the way: we are using numbers in the name purely for fun.

```
function inthehat thing1 thing2
    pass
    fig
```

is a valid definition. Let's have it **print** the number of characters in each parameter.


```
function inthehat thing1 thing2
    q 34  chr
    now q  plus thing1  plus q  plus " has " prints
    now thing1  len  str plus " characters." print

    now q  plus thing2  plus q  plus " has " prints
    now thing2  len  str plus " characters." print
    fig
```

and let's call the function, with two strings:

```
now  inthehat "hello" "mike"
```

outputs this:

```
"hello" has 5 characters.
"mike" has 4 characters.
```

One of the fun things about this code is it adds ascii 34 (a computer code for a double quote) to the string, so the output can display the string in quotes. That would be a useful function actually:

```
function quote qstring
q 34  chr
withquotes q  plus  qstring  plus q  print
fig

now "this will be printed in quotes"  quote now
```

It works perfectly, but all it does is build the quotes into the string and print it. Wouldn't it be cool if like other built-in functions, it could change the value of the main variable?

Well, that's what **return** is for. Let's just change one line:

```
function quote qstring
q 34  chr
withquotes q plus qstring plus q  return withquotes
fig
```

Now we have a function we can use anywhere in our program-- so long as it's after the function definition. Let's fix up our other function with it. It does the same as before, with cleaner lines:

```
function quote qstring
q 34  chr
withquotes q plus qstring plus q  return withquotes
fig

function inthehat thing1 thing2
    now  quote thing1  plus " has " prints
    now thing1  len  str plus " characters." print

    now  quote thing2  plus " has " prints
    now thing2  len  str plus " characters." print
    fig
```


And call it:

```
now  inthehat "hello" "mike"
```

"hello" has 5 characters.
"mike" has 4 characters.

And it works! Our function uses the **len** function (built in) to figure out how many characters the strings have-- len will also find the length of an array, but not a numeric. So if you do this:

```
now  inthehat "hello" 5
```

Your function will not work. You can fix this by putting the line that has **len** in a **try** / **except** / **resume** conditional block, and deciding what to say if it trips the **except** section.

You can also reword the **" characters."** string so that it says something like **" characters (or array elements)."** That's up to your preference obviously.

Hopefully from these examples you can practice writing and editing functions. They are very powerful: half the job of creating your own programming language could be writing functions. Fig has more functions built-in, so let's see them:

```
x  y  lcase  # copy y to x and make all-lowercase

x  y  ucase  # copy y to x and make all-uppercase

x  y  str  # copy y to x and convert num to string

x  "dir"  shell  # run commands in bash/sh or dos

x  "hello"  asc  # convert first character in a
string to numeric ascii code

x  "50.537"  val # convert string number to numeric

x  "hello there"  len  # length of string or array

x  52  not  # return -1 for zero and 0 for non-zero

x  "  space from left"  ltrim # cut lefthand space

x  "space from right  " rtrim # cut righthand space

x  10  chr  # convert integer to ascii / unicode
```



```
x  "dir"  arrshell  # load an array with cli output

x  arreverse  # reverse the order of an array

x  y  reverse  # copy string y to x and reverse

x  arrsort  # sort an array

x "hello"  left 2  # get leftmost 2 characters

x "hello"  right 2  # get rightmost 2 characters

x  arrget rr 5  # set x to 5th item of array rr

rr arrset 5 "hello" # set 5th item in rr to "hello"

x  y  mid 5 1  # copy y to x, and set x to a
range/section of 1 character(s) or item(s) starting
with the 5th. (works on strings and arrays.)

p  string 12 104  # string of 12 x  ascii 104

p  string 12 "h"  # string of 12 x  "h"
```

```
x  split "hello" "e"  # split string by "e" into array ['h', 'llo']


e  join x "a"  # join array using "a" in between items
```

Splitting by "e" and joining by "a" will change all instances of "e" to "a" in a string. You can create a function called **replace** like this:

```
function replace changewhat chfrom chto
    p  split changewhat chfrom  join p chto
    now  return p
    fig
```

then you can call it this way:

```
phrase "hello there"  replace phrase "he" "a"


x  instr "hello" "e"   # finds the first instance of "e" in "hello" and returns the position; which in this case is 2 (0 if not found.)


x  "/"  chdir  # changes the folder the program is working in.
```



```
now  end  # quits the program (closes files too)


now  system  # exactly the same as end


now  swap x y  # switches the values of x and y


now  get parametername  # earlier versions of fig required this to copy parameters inside functions; it is no longer required, but can still be used to copy variables into the main variable.
```

Here are a couple more functions you can use; Basic and Python (therefore fig) use radians for angles in trig functions. Perhaps you would like to use degrees:


```
function degrees2radians dg
    pi 1  atn  times 4
    rd dg  times pi  divby 180  return rd
    fig


function radians2degrees rd
    pi 1  atn  times 4
    dg rd  times 180  divby pi  return dg
    fig
```

Chapter 8: A little Python

As far as coding goes, you can write pretty much anything in fig you want to. It may not be the fastest language, or have a feature for everything, but you can extend it using Python.

Python is a great second language to learn, and is still probably easier than Javascript (although Javascript will run in your browser.)

If you have no interest in Python yet, you don't actually need this chapter. Think of it as "extra credit," but definitely don't worry if Python seems too complicated.

A few months of coding in any language should make learning Python easier, and there are better intros to the language than this one; this one is just made for people comfortable with fig.

If this chapter only started with what makes Python different than fig, you might think Python is either pure genius (it basically is) or that it makes things more difficult than necessary (it sometimes does.)

It should be noted that Python is a little closer to what you should expect from most languages than fig is. Fig was designed specifically to round off (some might even say cut) as many corners as

possible. Today, most languages are case-sensitive, Python has to be indented **just so**, and while fig has a "main variable" at the beginning of most lines, Python requires that you at least set every variable to something before you can reference it.

Fig COULD automatically return 0 for any variable referenced, even if not used already. Since most modern languages don't do this (and even have a good argument against it) Fig uses its main variable concept instead, which has the following features:

* It makes it obvious what variables are used in a program

* It zeroes unused variables in a way that is less work than Python, but relatively compatible with most modern languages-- it is minimalist yet still explicit.

* It works like a "named pipe," (a concept in Unix-like operating systems) where a number of built-in functions can reference and change the contents of a stream of data implicitly:

```
x arropen "file.txt" | join x " " | ucase | print
```

works kind of like this command in GNU/Linux:

```
x=$(cat file.txt | sed 's/\n/\ /g' | tr a-z A-Z) ;
echo $x
```

The pipes: **|** in fig don't change the way that fig works, they are simply a substitute command separator for Python's "**;**" or Basic's "**:**".

Fig is therefore also a simple (sort of) introduction to coding in Bash. Although it barely is the case, the main variable functionality does transition a little to using pipes in Bash.

But getting back to Python, you must name (and set) a variable before you can reference (or use) it:

```
x = 0 ; print(x)  # Python uses hashes for comments
```

because fig allows a certain amount of decorative punctuation, this will work in fig:

```
x = 0 ; print  # fig uses hashes for comments too
```

but the **x** after **print** is implied. In Python, the semicolon **;** between commands and the equals **=** between variable name and value are required. And of course, every function, like print(), requires the variable to be named.

```
x = 0 ; print (x) ; print (x)  # Python

x   0   print       print      # fig
```


another thing worth mentioning about Python is that it comes in two major versions: 2 and 3, which are both still in use. A lot of professionals still prefer Python 2, as do I; fig was conceived as a Python 2 project, and I would love to make fig a project for Python 2 in the long run.

Fig from version 3.0 onward runs in Python 3. More about this momentarily.

In some ways, Python 2 is more friendly and flexible. The biggest difference is string handling, and the **print** command. In Python 2, **print** works like this:

```
print x
print x,
print x, y, z
```

In Python 3, **print** works like this:

```
print(x)
print(x, end=' ')
print(x, y, z)
```

Ok, so what's the big deal then? Well it doesn't stop there, if you want to save or load a file, or print certain characters, or read from stdin-- all stuff Python 2 makes trivial that's more of a hassle in Python 3.

Fig would still be a Python 2 project, except the Python Foundation was supposed to drop support for it in 2015, and has only extended it to 2020. I still hope that the many people who prefer 2 will fork the language, perhaps calling it "boa" or something.

At that point, I would most likely change fig back to Python 2 (because transitioning is a feature, and I think 2 is better for education.) But in the meantime, fig 3.0 onward is a Python 3 project. If someone is interested enough in Python 2.9 (or earlier) to fork or maintain it, they should feel (they are) free to do so; I would encourage it.

Fig was made with the hopes that more people would understand and even write programming languages, and a programming language made from (any version of) fig would be a very cool thing to find.

I'm focused on Python 3-- reluctantly-- because it may really be the version of Python we are all stuck with sooner or later. It's too bad.

Anyway, a fig program that works in 2.9 will probably work the same in fig 3.x, but eventually fig may have features (**command** is one) that aren't available in 2.9.

As for inline Python, the version of fig (2 vs. 3) is also the version of Python you must use. So fig 3.1 requires inline Python 3 syntax.

One of the largest differences between fig and Python is indentation. A lot of my fig examples I indent "Python-style" but in Python, you must indent each block of code: and you must unindent at the end of it. So this **for** loop in fig:

```
for x (1 10 2)
    now x print
    next
```

corresponds to the following loop in Python:

```
for x in range(1, 10 + 1, 2):
    now = x ; print(now)
```

There's no "next" in Python; the change in indentation is how it knows!

Nested loops in fig and Python:

```
for y (1 5 1)
    for x (1 10 2)
        now x prints " " prints
        now y print
        next
    next

for y in range(1, 5 + 1, 1):
    for x in range(1, 10 + 1, 2):
        now = x ; print(now, end=' ')
        now = y ; print(now)
```

In Python, some functions are built in (like **int()** and **print()** and **join()** for example) while others have to be **import**ed:

```
x 3.14  cos  print  # fig

from math import cos ; print cos(3.14) # Python
```

Just like defining a function, you only have to import cos once per program to use it as many times as you like. Another way to do the same thing:

```
import math ; print math.cos(3.14)
```

The advantage of doing it that way is that it imports the entire math library at once, so you don't have to keep importing each of its commands individually.

```
from sys import stdin
for p in stdin: ps += [p[:-1]]
return ps[:]
```

This is where fig's **arrstdin** comes from.

Fig's **function** is named for the Basic command, but:

```
def replace(phrase, chfrom, chto):
    phrase = phrase.split(chfrom)
    return chto.join(phrase)
```


in Python, **function** is called **def**.

Note also that **join x y** becomes y.join(x) and **split x y** becomes **x.split(y)** ...in python you can just do this, which is easier:

```
phrase = phrase.replace(x, y)
```

Block commands aside, most functions in fig start with **def fig**commandname in fig's Python source. So in the fig translator itself:

```
def figint(p): return int(p)
```

is the code in Python that performs **int** in fig. Sometimes the translation is that simple, and sometimes it isn't. From inline Python in fig:

```
x 12.5
python
    figcolortext(0, 1) ; fighighlight(0, 7)
    # fig's int function:
    print(figint(x) * 10)
fig
```

So not only can you use inline Python, you can use fig functions (some of them) inside inline Python. When all of it is inline Python, you can copy the fig functions and use them in a pure Python script.

Chapter 9: Debugging

Fig doesn't try to stop every Python error you might encounter. The goal of fig is to produce working Python code, and working Python code will return errors under certain circumstances.

If you want to, you can learn more about Python and this will make you better at debugging fig programs. However, if you get to know fig well enough, you can get pretty good at debugging code even without understanding Python errors.

First of all, you can stop quite a few errors just by putting your program between **try** and **except** commands, then putting **pass** between **except** and **resume**, like this:

```
try
    # your program code goes here
except
    pass
resume
```

THAT DOESN'T MEAN YOU REALLY WANT TO DO THIS!

Error messages can be extremely good clues towards what needs to be fixed. (Other times, not so much.) They may not "look good" but they don't put them in because they can't help it; they put them in so

that the coder-- and the user-- knows something (often easy to fix) is wrong.

Suppressing errors also suppresses fixes, but it rarely prevents the actual problems.

Then again, one of the better ways to see if a file exists is to open it and catch the error if it fails. (It's probably true.)

Things to consider when the program isn't working:

* **Are you working with strings or numerics?** (or arrays?) You can change one to another, but some commands and your own user-defined functions may require specifically strings or specifically numerics (**pset** even requires you convert with **int**.)

* **If you're using inline Python**, are the variables you're calling already used outside the Python code? (fig won't read variables from inline Python unless they were used in the fig code first.)

* **Are all the variables spelled the same way?** You can use a misspelled variable, like **lyte**, but you obviously can't expect **lyte** and **lite** to have the same value unless you copy one to the other. You can however use **Lyte** and **lyTE** as the same thing.

* **Inside and outside the loop:** make certain that loops are closed with **next** or **wend** or **fig** (or whatever, so long as each loop is closed in the right place.)

* M**ake certain that each "shared" line starts with a main variable**. Fig will usually tell you when you forget about this:

```
error: shared-line function "print" cannot be used to start a line
error in line 1:
print
```


Chapter 10: Example Programs

You can make your own programs from the things you learned in this manual, or you can make changes to the examples in this chapter. Feel free to do whatever you want with the code in this chapter, it is yours (this entire manual is licensed CC0) for any kind of use/re-use that you would like.

Programs will be separated with comments like this:

```
##################################################

for y 0 15 1
    for x 0 15 1
        now colortext x y
        now highlight y
        now x hex right 1 prints colortext 7
        now highlight 0
        next
    now "" print
    next

##################################################

)() fish )()
these )() fish "do nothing at all" )()
)() really. # fig can understand this code; but the
fish #are ignored.

##################################################
```

```
################################################

while

    x "type in file to open - " prints lineinput
    ifequal x "quit"
        break
        fig

    p arropen x
    forin lines p
        words split lines " "

        forin z words
            x z sleep .15 print
            nextin

        nextin

    wend

################################################

forin items stdin            # pipe stdin into loop
now items ucase rtrim print  # print each uppercase
nextin

################################################

p "hello" len
python
    p = 1 / 3.0 + (p * (3 + 7 * 9) ** 5.7)
fig
z p print                                    # 77
```

```
################################################

z "rhino-05"
p "rhino-09"

ifless z p
x "yes, z is less than p" print
fig

################################################

while
    p : "enter 0 to continue, 1 to quit:" : print
    p : lineinput : val
    iftrue p
        break # stop looping
        fig
    wend

while
    text "type words to uppercase, " prints
    text "or 'quit' to end" print
    text lineinput ucase print lcase
    ifequal text "quit"
        break
        fig
    wend

################################################
```

```
################################################

r : 200
c
x : 640 : divby 2
y : 480 : divby 2

for p -3.14 3.14 .628
for pr -3.14 3.14 .628

dx  : p  : cos : times r
dy  : p  : sin : times r
dxr : pr : cos : times r
dyr : pr : sin : times r

z  : c : plus 1 : swap z c

xx : x : plus 5 : plus dx  : int
yy : y : plus 5 : plus dy  : int
x2 : x : plus 5 : plus dxr : int
y2 : y : plus 5 : plus dyr : int

cc : c : mod 5 : plus 10
z  : line xx yy x2 y2 cc

z : sleep .005

next
next

z : lineinput

################################################
```



```
################################################

p = "hello"
x == "hey"
a = "yes" : arr

ifequal "hello" == p
    a : print
    fig
ifequal "hey" = x
    a : print
    fig
ifequal p == x
    pass
    else
    n == "no" : print
    fig

################################################

# this is a working fig program
function printhello
    p cls # clear screen
    p "hello world" print
    fig
python
    # inline python feature
    for p in range(5):
        printhello()
    fig


################################################
```

```
################################################

function htext x y pal
    p      "0000111000000011110000" arr
    p plus "0011000110000110001100"
    p plus "0100000001010000000010"
    p plus "1000000000100000000001"
    p plus "0100000000000000000010"
    p plus "0011000000000000001100"
    p plus "0000100000000000010000"
    p plus "0000011000000001100000"
    p plus "0000000100000010000000"
    p plus "0000000011011000000000"
    p plus "0000000000100000000000"
    plen p  len
    for r 1 plen 1
        hrow arrget p r #### get line
        now ""  highlight 0  print
        vert y  plus r  locate vert, x
        hlen hrow  len
        for c 1 hlen 1
            ch  arrget hrow c val
            xc x  plus c  locate vert, xc
            iftrue ch
                now  highlight pal " " prints
                fig
            next
        next
    fig
try
    while
        palchoice randint 1 3  sleep .25
        pal "457"  mid palchoice 1  val
        x randint 5 55                          # 81
        y randint 2 12  htext x y pal
        wend
except
    pass
    resume
now "" colotext 7 highlight 0  print

################################################

c = 0
textmode
for radius (1, 200, .157)
x = 320
y = 240

for b (-3.14, 3.14, .314)
d = c plus .157 : swap d, c
xc = b plus c : cos
yc = b plus c : sin
col = xc times radius plus x
row = yc times radius plus y
colr = b times 3 : abs : mod 2 : int : times 11
plus 4
col2 = col divby 8 : int : plus 34
row2 = row divby 16 minus 2 : int : plus 1
xx = col2 minus 1
x2 = col2 minus 0
z line(xx, row2, x2, row2), colr
next

next
p : colortext 7 : locate 1, 1

################################################ 82
```