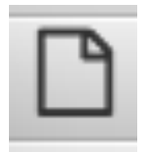
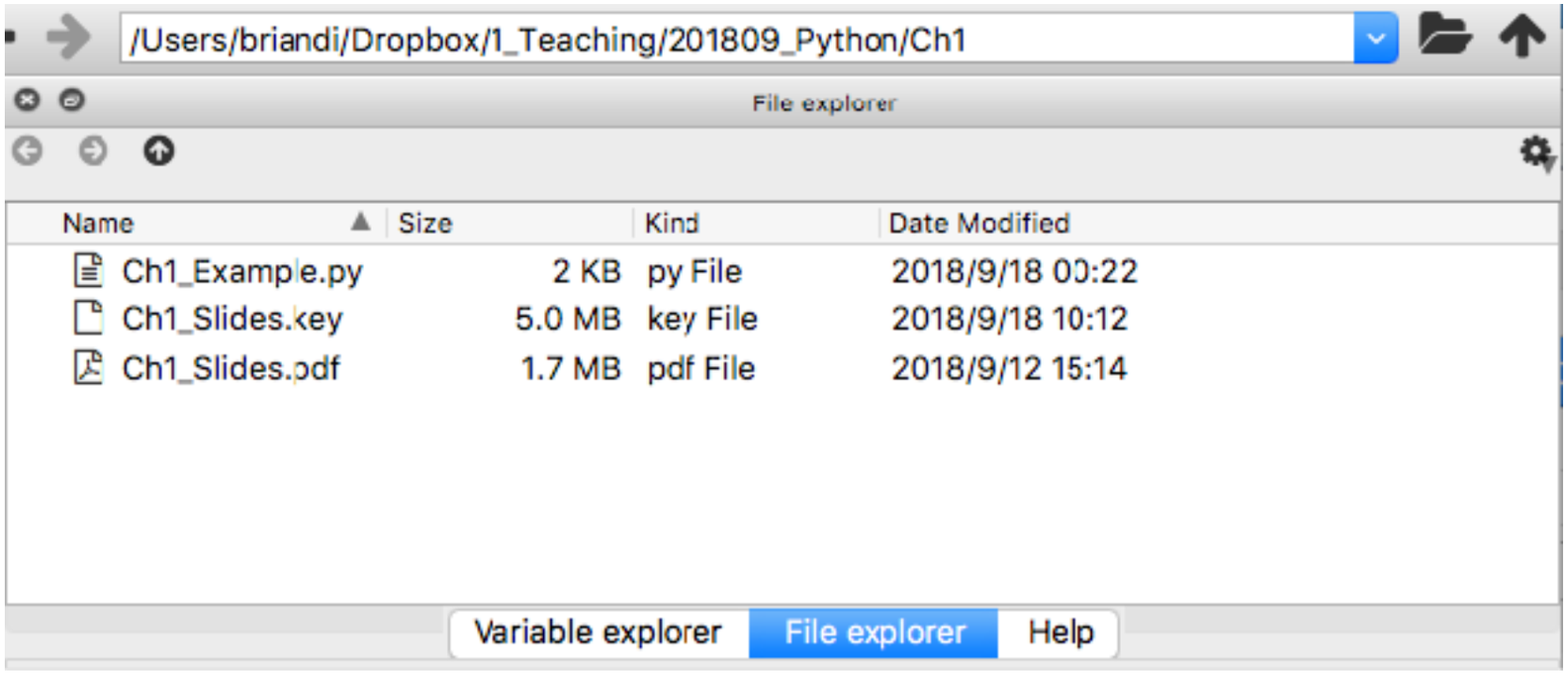
Introduction to Computer Science

Program Flow

鄒年棣 (Nien-Ti Tsou)

Running the Python by script

- An py-file is a plain text file containing Python commands and saved with the filename extension .py
- Create a new py-file 
- Script (腳本、劇本): You can write many commands, save it and execute it by click “play” button. 
- Working directory: Python will look for files in the folder you specified.



Running the Python by script

Spyder (Python 3

Editor - /Users/briandi/Desktop/untitled6.py

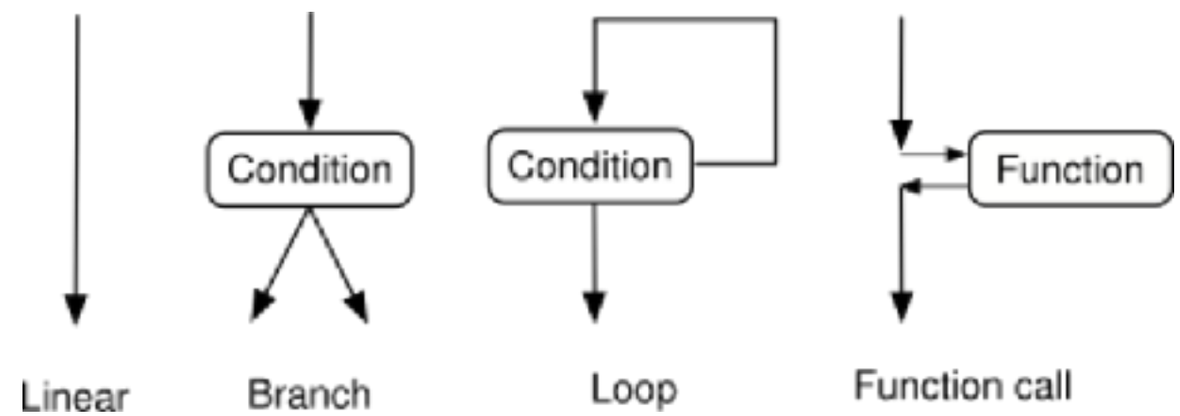
untitled6.py*

```
Onestr = 'these are double quotes: ".." '
Morestr = """ Many lines 1~
    Many lines 2~
Many lines 3~"""
print(Onestr)
print(Morestr)
```

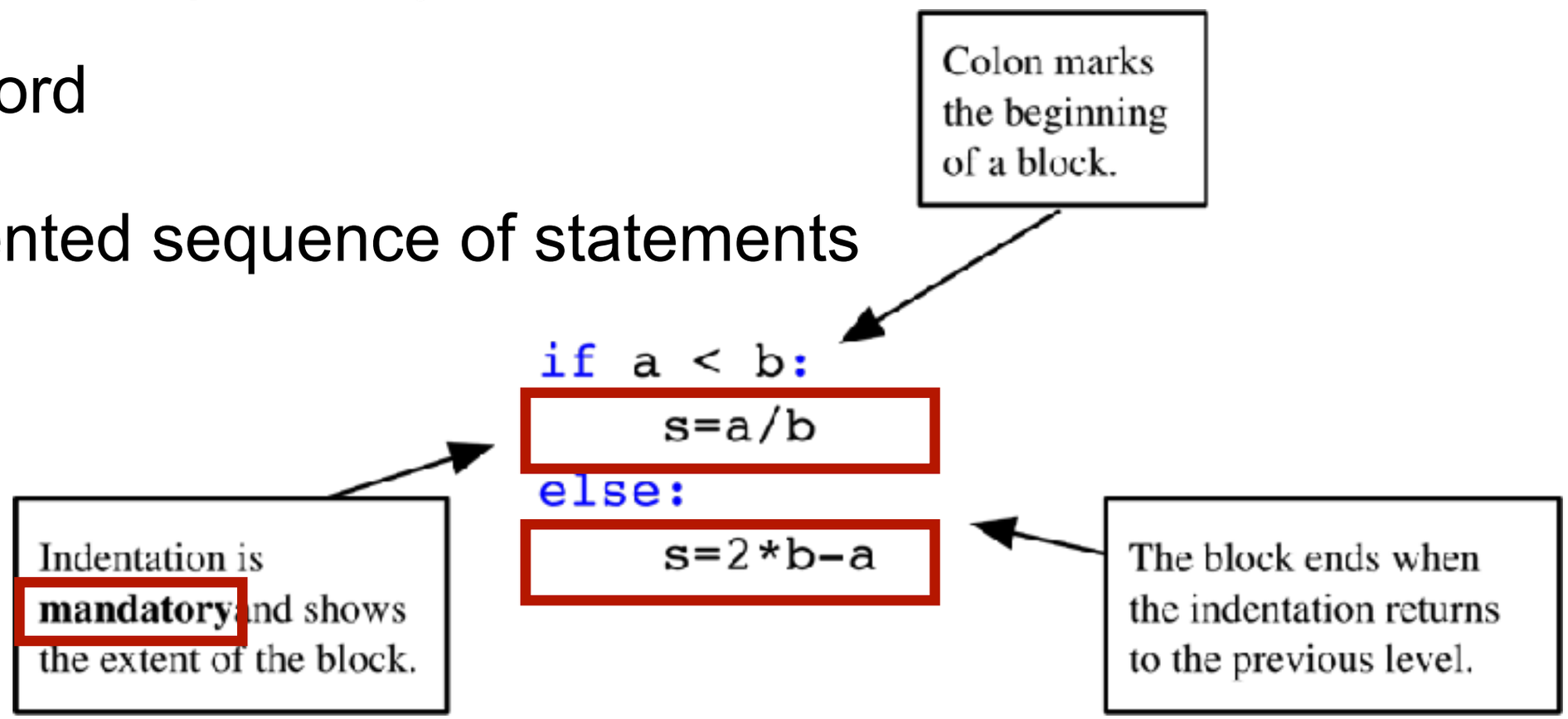
```
In [9]: runfile('/Users/briandi/Desktop/
untitled6.py', wdir='/Users/briandi/
Desktop')
these are double quotes: ".."
 Many lines 1~
    Many lines 2~
Many lines 3~
```

Program and program flow

- A program is a sequence of statements that are executed in a top-down order, except:



- Python uses a special syntax to mark blocks of statements:
  - a keyword
  - a colon
  - an indented sequence of statements



Boolean expressions

- An expression for `True` or `False`.
  - Equal, ==
  - Not equal, `!=`
  - Less than, Less than or equal to, < , <=
  - Greater than, Greater than or equal to, > , >=
- Combines with `or` and `and`.
  - `not` , gives the logical negation of the expression that follows
  - Precedence rules

Low
- 1. or
- 2. and
- 3. not x
- 4. <, <=, >, >=, !=, ==

High

```
2 >= 4
2 < 3 < 4
2 < 3 and 3 < 2
2 != 3 < 4 or False
2 <= 2 and 2 >= 2
not 2 == 3
not False or True and False
((not False) or True) and False
```

Boolean expressions

- An expression for `True` or `False`.
  - Equal, ==
  - Not equal, `!=`
  - Less than, Less than or equal to, < , <=
  - Greater than, Greater than or equal to, > , >=
- Combines with `or` and `and`.
- `not` , gives the logical negation of the expression that follows
- Precedence rules

Low
- 1. or
- 2. and
- 3. not x
- 4. <, <=, >, >=, !=, ==

High

```
2 >= 4
2 < 3 < 4
2 < 3 and 3 < 2
2 != 3 < 4 or False
2 <= 2 and 2 >= 2
not 2 == 3
not False or True and False
((not False) or True) and False
```

```
False
True
False
True
True
True
True
False
```

Boolean expressions

- An expression for `True` or `False`.
  - Equal, ==
  - Not equal, `!=`
  - Less than, Less than or equal to, < , <=
  - Greater than, Greater than or equal to, > , >=
- Combines with `or` and `and`.
- `not` , gives the logical negation of the expression that follows
- Precedence rules

Low
- 1. or
- 2. and
- 3. not x
- 4. <, <=, >, >=, !=, ==

High

```
2 >= 4
2 < 3 < 4
2 < 3 and 3 < 2
2 != 3 < 4 or False
2 <= 2 and 2 >= 2
not 2 == 3
True or True and False
((not False) or True) and False
```

```
False
True
False
True
True
True
True
False
```

Boolean expressions

- An expression for `True` or `False`.
  - Equal, ==
  - Not equal, `!=`
  - Less than, Less than or equal to, < , <=
  - Greater than, Greater than or equal to, > , >=
- Combines with `or` and `and`.
  - `not` , gives the logical negation of the expression that follows
  - Precedence rules

Low
- 1. or
- 2. and
- 3. not x
- 4. <, <=, >, >=, !=, ==

High

```
2 >= 4
2 < 3 < 4
2 < 3 and 3 < 2
2 != 3 < 4 or False
2 <= 2 and 2 >= 2
not 2 == 3
   True    or      False
((not False) or True) and False
```

```
False
True
False
True
True
True
True
False
```

Booleans

- Boolean is a datatype named after George Boole (1815-1864). The main use of this type is in logical expressions

```
a = True
b = 30 > 45
print(a,b)
```

- Note that the `and` operator is implicitly chained in the following Boolean expressions:

```
a < b < c# same as: a < b and b < c
a == b == c # same as: a == b and b == c
```

Booleans

- Boolean is a datatype named after George Boole (1815-1864). The main use of this type is in logical expressions

```
a = True
b = 30 > 45
print(a,b)
```

```
True False
```

- Note that the `and` operator is implicitly chained in the following Boolean expressions:

```
a < b < c# same as: a < b and b < c
a == b == c # same as: a == b and b == c
```

Boolean casting

- The built-in function `bool` converts objects to Booleans
- Note that most objects are cast to `True`
- The objects which cast to `False` are `0`, the empty list, the empty string, the empty tuple, or the empty array.

```
bool([])
bool(0)
bool(' ')
bool('')
bool('hello')
bool(1.2)
```

Bool	False	True
string	''	'not empty'
number	0	≠
list	[]	[...] (not empty)
tuple	()	(..,..) (not empty)
array	array([])	array([a]) (a ≠ 0)
array	array([0])	
array	Exception raised if array contains more than one element	

`tuple` and `array` will be explained later

Boolean casting

- The built-in function `bool` converts objects to Booleans
- Note that most objects are cast to `True`
- The objects which cast to `False` are `0`, the empty list, the empty string, the empty tuple, or the empty array.

```
bool([])
bool(0)
bool(' ')
bool('')
bool('hello')
bool(1.2)
```

```
False
False
True
False
True
True
```

Bool	False	True
string	''	'not empty'
number	0	≠
list	[]	[...] (not empty)
tuple	()	(..,..) (not empty)
array	array([])	array([a]) (a ≠ 0)
array	array([0])	
array	Exception raised if array contains more than one element	

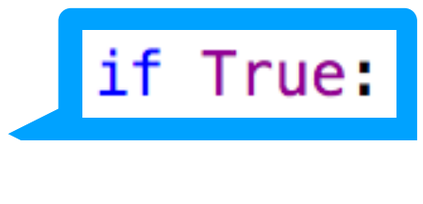
`tuple` and `array` will be explained later

Conditional statements in our daily life.

```
# This is not a script for python
if the weather is sunny:
    We go out and play!
else if it is windy and not rainy:
    We go out with a jacket!
else if it is just drizzling:
    We go out with an umbrella!
else:
    Stay at home…
```

Conditional statements: `if elif else`

- A conditional statement delimits a block that will be executed if the condition is true.
- An optional block, started with the keyword `elif else` will be executed if the condition is not fulfilled

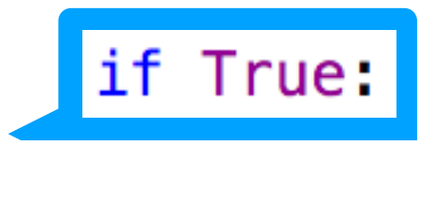



```
x=3
if x>=0:
    print('1. Absolute value is: ')
    abs_x = x
elif x<0:
    print('2. Absolute value is: ')
    abs_x = -x
else:
    print('3. Something wrong')
print(abs_x)
```

outside the `if elif else` blocks

Conditional statements: `if elif else`

- A conditional statement delimits a block that will be executed if the condition is true.
- An optional block, started with the keyword `elif else` will be executed if the condition is not fulfilled




```
x=3
if x>=0:
    print('1. Absolute value is: ')
    abs_x = x
elif x<0:
    print('2. Absolute value is: ')
    abs_x = -x
else:
    print('3. Something wrong')
print(abs_x)
```

```
1. Absolute value is:
3
```

outside the `if elif else` blocks

Conditional statements: `if elif else`

```
day = 2
if (day>5) and (day<8):
    print('Weekend!')
elif (day>1) and (day<6):
    print('Weekday.')
elif day==1:
    print('Blue Monday.')
else:
    print('Something wrong.')
```

You can have as many `elif` as you want.

Conditional statements: `if elif else`

```
day = 2
if (day>5) and (day<8):
    print('Weekend!')
elif (day>1) and (day<6):
    print('Weekday.')
elif day==1:
    print('Blue Monday.')
else:
    print('Something wrong.')
```

```
Weekday.
```

You can have as many `elif` as you want.

Is it correct?

```
if x >= 0:
    print(x)
else x < 0:
    print(-x)
```

Is it correct?

```
if x >= 0:
    print(x
else x < 0:
    print(-x)
```

```
    else x < 0:
         ^
SyntaxError: invalid syntax
```

Quiz!

Please write a program. Define a variable named: grade, and write conditional statements to output: the corresponding letter A, B, C, D or F

Note:
A: >= 90
B: 80~89.999999999
C: 70~79.999999999
D: 60~69.999999999
F: < 60

Automatic Boolean casting

- Using `if`, `or`, `and`, and `not` statement with a non-Boolean type will cast it to a Boolean automatically.

```
if a:
    ...
if bool(a): # exactly the same as above
    ...
```

- A typical example is testing whether a list is empty:

```
L = []
if L:
    print("list not empty")
else:
    print("list is empty")
```

- Another example is testing whether a number is odd:

```
n = 4
if n % 2:
    print("n is odd")
else:
    print("n is even")
```

% for the modulo operation (求餘數)

Automatic Boolean casting

- Using `if`, `or`, `and`, and `not` statement with a non-Boolean type will cast it to a Boolean automatically.

```
if a:
    ...
if bool(a): # exactly the same as above
    ...
```

- A typical example is testing whether a list is empty:

```
L = []
if L:
    print("list not empty")
else:
    print("list is empty")
```

```
list is empty
```

- Another example is testing whether a number is odd:

```
n = 4
if n % 2:
    print("n is odd")
else:
    print("n is even")
```

% for the modulo operation (求餘數)

Automatic Boolean casting

- Using `if`, `or`, `and`, and `not` statement with a non-Boolean type will cast it to a Boolean automatically.

```
if a:
    ...
if bool(a): # exactly the same as above
    ...
```

- A typical example is testing whether a list is empty:

```
L = []
if L:
    print("list not empty")
else:
    print("list is empty")
```

```
list is empty
```

- Another example is testing whether a number is odd:

```
n = 4
if n % 2:
    print("n is odd")
else:
    print("n is even")
```

% for the modulo operation (求餘數)

```
n is even
```

Arithmetic Operations

Operator	Name	Description
a + b	Addition	Sum of a and b
a - b	Subtraction	Difference of a and b
a * b	Multiplication	Product of a and b
a / b	True division	Quotient of a and b
a // b	Floor division	Quotient of a and b, removing fractional parts
a % b	Modulus	Remainder after division of a by b
a ** b	Exponentiation	a raised to the power of b
-a	Negation	The negative of a
+a	Unary plus	a unchanged (rarely used)

```
# True division
print(11 / 2)
# Floor division
print(11 // 2)
```

Arithmetic Operations

Operator	Name	Description
a + b	Addition	Sum of a and b
a - b	Subtraction	Difference of a and b
a * b	Multiplication	Product of a and b
a / b	True division	Quotient of a and b
a // b	Floor division	Quotient of a and b, removing fractional parts
a % b	Modulus	Remainder after division of a by b
a ** b	Exponentiation	a raised to the power of b
-a	Negation	The negative of a
+a	Unary plus	a unchanged (rarely used)

```
# True division
print(11 / 2)
# Floor division
print(11 // 2)
```

```
5.5
5
```

Repeating statements with loops: `for`

- Loops are used to repetitively execute a sequence of statements while changing a variable from iteration to iteration.

- This variable is called the index variable. It is successively assigned to the elements of a list.

```
L = [1, 2, 10]
for s in L:
    print(s * 2)
print(s)
```

outside the for block

```
L = [1, 2, 10]
for s in range(3):
    print(L[s] * 2)
print(s)
```

Repeating statements with loops: `for`

- Loops are used to repetitively execute a sequence of statements while changing a variable from iteration to iteration.

- This variable is called the index variable. It is successively assigned to the elements of a list.

```
L = [1, 2, 10]
for s in L:
    print(s * 2)
print(s)
```

outside the for block

```
2
4
20
10
```

```
L = [1, 2, 10]
for s in range(3):
    print(L[s] * 2)
print(s)
```

Repeating statements with loops: `for`

- Loops are used to repetitively execute a sequence of statements while changing a variable from iteration to iteration.
- This variable is called the index variable. It is successively assigned to the elements of a list.

```
L = [1, 2, 10]
for s in L:
    print(s * 2)
print(s)
```

outside the for block

```
2
4
20
10
```

```
L = [1, 2, 10]
for s in range(3):
    print(L[s] * 2)
print(s)
```

```
2
4
20
2
```

Repeating statements with loops: `for`

```
In [34]: list(range(11))
Out[34]:
```

```
Sum = 0
for x in list(range(11)):
    Sum = Sum +x
print(Sum)
```

```
print(x)
```

```
for x in range(11):
    Sum = Sum +x
print(Sum)
print(x)
```

Note!! The first letter of `Sum` is capital. `sum` is a built-in function

Repeating statements with loops: `for`

```
In [34]: list(range(11))
Out[34]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
Sum = 0
for x in list(range(11)):
    Sum = Sum +x
print(Sum)
```

```
print(x)
```

```
for x in range(11):
    Sum = Sum +x
print(Sum)
print(x)
```

Note!! The first letter of `Sum` is capital. `sum` is a built-in function

Repeating statements with loops: `for`

```
In [34]: list(range(11))
Out[34]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
Sum = 0
for x in list(range(11)):
    Sum = Sum +x
print(Sum)
```

```
55
```

```
print(x)
```

```
for x in range(11):
    Sum = Sum +x
print(Sum)
print(x)
```

Note!! The first letter of `Sum` is capital. `sum` is a built-in function

Repeating statements with loops: `for`

```
In [34]: list(range(11))
Out[34]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
Sum = 0
for x in list(range(11)):
    Sum = Sum +x
print(Sum)
```

```
55
```

```
print(x)
```

```
10
```

```
for x in range(11):
    Sum = Sum +x
print(Sum)
print(x)
```

Note!! The first letter of `Sum` is capital. `sum` is a built-in function

Repeating statements with loops: `for`

```
In [34]: list(range(11))
Out[34]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
Sum = 0
for x in list(range(11)):
    Sum = Sum +x
print(Sum)
```

```
55
```

```
print(x)
```

```
10
```

```
for x in range(11):
    Sum = Sum +x
print(Sum)
print(x)
```

```
110
10
```

Note!! The first letter of `Sum` is capital. `sum` is a built-in function

Repeating statements with loops: `for`

- Find the maximum value in a list step by step.

```
A = [ 6, 12, 6, 91, 13, 6]
theMax = A[1]  # set initial max value
for x in A :# iterate through A
    if x > theMax: # test each element
        theMax=x
print(theMax)
```

91

This example is for practicing only. Python provide built-in function `max(A)`.

Repeating statements with loops: `for`

- Find how many 50 in the list below step by step.

```
L = [50, 2, 22, 88, 97, 29, 123,
     37, 45, 50, 99, 53,
     21, 11, 23, 45, 238, 516,
     33, 56, 868, 50, 24, 99, 778, 99]
```

```
L = [50, 2, 22, 88, 97, 29, 123,
     37, 45, 50, 99, 53,
     21, 11, 23, 45, 238, 516,
     33, 56, 868, 50, 24, 99, 778, 99]

Count = 0
Target = 50
for i in L:
    if i == Target:
        Count += 1
print(Count)
```

```
3
```

Repeating statements with double loops

- Make a multiplication table

```
for i in range(     ):
    print('====',i,'====')
    for j in range(     ):
        print(i,'*',j,'=',i*j)
print('===========')
```

```
==== 1 ====
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
1 * 7 = 7
1 * 8 = 8
1 * 9 = 9
==== 2 ====
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6

.
.
.

8 * 8 = 64
8 * 9 = 72
==== 9 ====
9 * 1 = 9
9 * 2 = 18
9 * 3 = 27
9 * 4 = 36
9 * 5 = 45
9 * 6 = 54
9 * 7 = 63
9 * 8 = 72
9 * 9 = 81
===========
```

Repeating statements with double loops

- Make a multiplication table

```
for i in range(1,10):
    print('====',i,'====')
    for j in range(     ):
        print(i,'*',j,'=',i*j)
print('===========')
```

```
==== 1 ====
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
1 * 7 = 7
1 * 8 = 8
1 * 9 = 9
==== 2 ====
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6

.
.
.

8 * 8 = 64
8 * 9 = 72
==== 9 ====
9 * 1 = 9
9 * 2 = 18
9 * 3 = 27
9 * 4 = 36
9 * 5 = 45
9 * 6 = 54
9 * 7 = 63
9 * 8 = 72
9 * 9 = 81
===========
```

Repeating statements with double loops

- Make a multiplication table

```
for i in range(1,10):
    print('====',i,'====')
    for j in range(1,10):
        print(i,'*',j,'=',i*j)
print('===========')
```

```
==== 1 ====
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
1 * 7 = 7
1 * 8 = 8
1 * 9 = 9
==== 2 ====
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6

.
.
.

8 * 8 = 64
8 * 9 = 72
==== 9 ====
9 * 1 = 9
9 * 2 = 18
9 * 3 = 27
9 * 4 = 36
9 * 5 = 45
9 * 6 = 54
9 * 7 = 63
9 * 8 = 72
9 * 9 = 81
===========
```

Break and else

- The `for` statement has two important keywords: `break` and `else`. `break` quits the for loop even if the list we are iterating is not exhausted.

```
for x in x_values:
    print(x)
    if x > threshold:
        break
print(x)
```

- The finalizing `else` checks whether the for loop was broken with the `break` keyword. If it was not broken, the block following the `else` keyword is executed:

```
x_values = list(range(11))
threshold = 100
for x in x_values:
    print(x)
    if x > threshold:
        break
else:
    print("all the x are below the threshold")
```

```
0
1
2
3
4
5
6
7
8
9
10
All the x are below the
threshold
```

Break and else

- The `for` statement has two important keywords: `break` and `else`. `break` quits the for loop even if the list we are iterating is not exhausted.

```
x_values = list(range(11))
threshold = 3.5
for x in x_values:
    print(x)
    if x > threshold:
        break
print(x)
```

- The finalizing `else` checks whether the for loop was broken with the `break` keyword. If it was not broken, the block following the `else` keyword is executed:

```
x_values = list(range(11))
threshold = 100
for x in x_values:
    print(x)
    if x > threshold:
        break
else:
    print("all the x are below the threshold")
```

```
0
1
2
3
4
5
6
7
8
9
10
All the x are below the
threshold
```

Break and else

- The `for` statement has two important keywords: `break` and `else`. `break` quits the for loop even if the list we are iterating is not exhausted.

```
x_values = list(range(11))
threshold = 3.5
for x in x_values:
    print(x)
    if x > threshold:
        break
print(x)
```

```
0
1
2
3
4
4
```

- The finalizing `else` checks whether the for loop was broken with the `break` keyword. If it was not broken, the block following the `else` keyword is executed:

```
x_values = list(range(11))
threshold = 100
for x in x_values:
    print(x)
    if x > threshold:
        break
else:
    print("all the x are below the threshold")
```

```
0
1
2
3
4
5
6
7
8
9
10
All the x are below the
threshold
```

Question!

```
x_values = list(range(11))
threshold =  ?
for x in x_values:
    print(x)
    if x >threshold:
        break
else:
    print("All the x are below the threshold")
```

What is the smallest threshold to get the following results?

```
0
1
2
3
4
5
6
7
8
9
10
```

Ans:

```
0
1
2
3
4
5
6
7
8
9
10
All the x are below the
threshold
```

Ans:

Question!

```
x_values = list(range(11))
threshold =  ?
for x in x_values:
    print(x)
    if x >threshold:
        break
else:
    print("All the x are below the threshold")
```

What is the smallest threshold to get the following results?

```
0
1
2
3
4
5
6
7
8
9
10
```

Ans: 9

```
0
1
2
3
4
5
6
7
8
9
10
All the x are below the
threshold
```

Ans:

Question!

```
x_values = list(range(11))
threshold =  ?
for x in x_values:
    print(x)
    if x >threshold:
        break
else:
    print("All the x are below the threshold")
```

What is the smallest threshold to get the following results?

```
0
1
2
3
4
5
6
7
8
9
10
```

Ans: 9

```
0
1
2
3
4
5
6
7
8
9
10
All the x are below the
threshold
```

Ans: 10

The `while` loop

- Infinite iterations are obtained with a `while` loop, or by recursion (will be explained in the future, hopefully).
- The `while` loop may be used to repeat a code block until a condition is fulfilled. The danger of this is: the code may be trapped in an infinite loop if the condition is never fulfilled.
- The number of run in For loops is set before the run.
- The number of run in While loops is set dynamically during the run.

```
Sum = 0
i = 0
while i<11:
    Sum += i
    i += 1
Sum
```

```
55
```

The `while` loop

```
x = 6
n = 0
while abs(x) > 1:
    x = x/2
    n = n + 1
    if n > 50:
        break
print('x =',x,'n=',n)
```

```
x = 0.75 n= 3
```

What is the initial value of *x* which makes the loop to exit with the command "`break`"?

An infinite loop

```
x=1
while x:
    print('x')
```

`CTRL + C` in case you run into an infinite loop

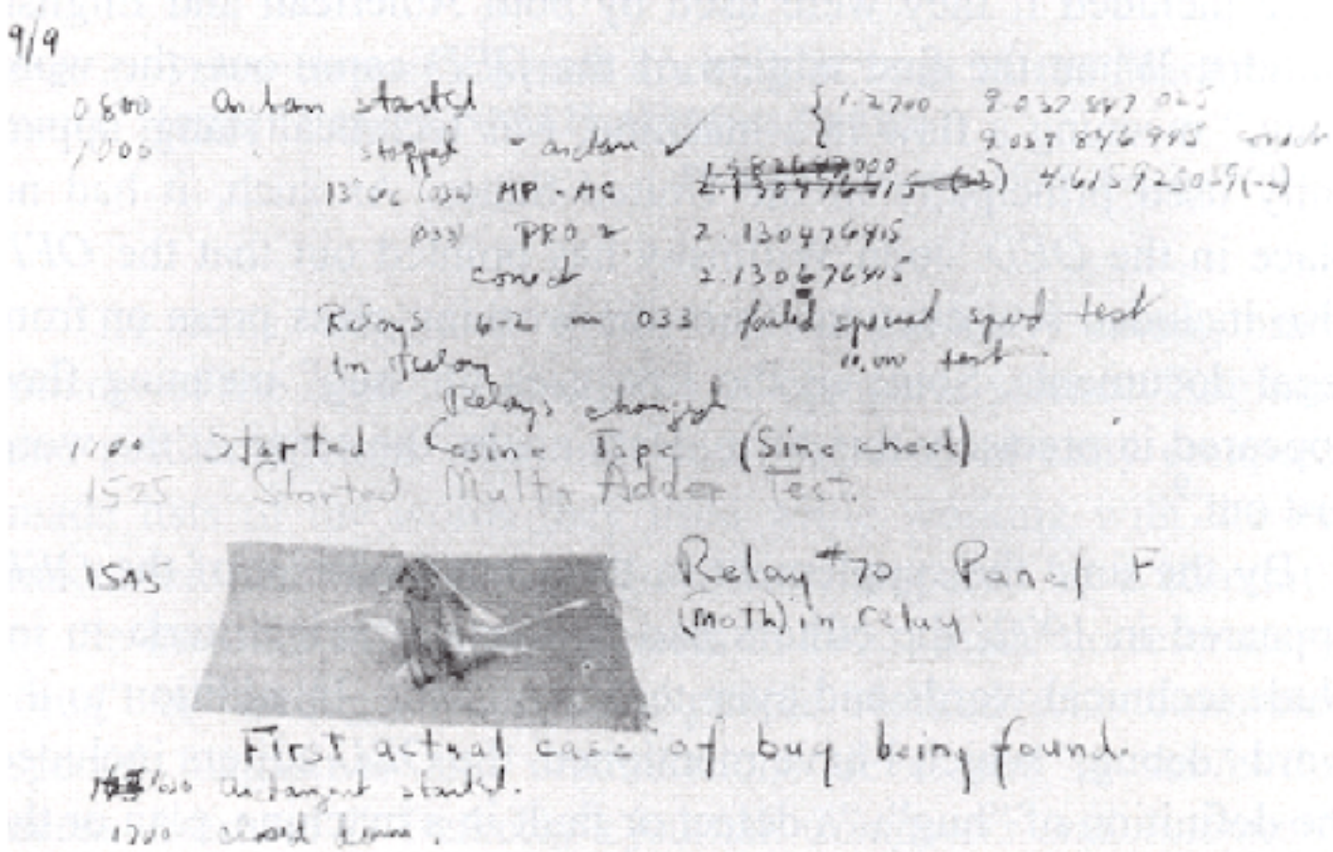
```
KeyboardInterrupt
```

Debug

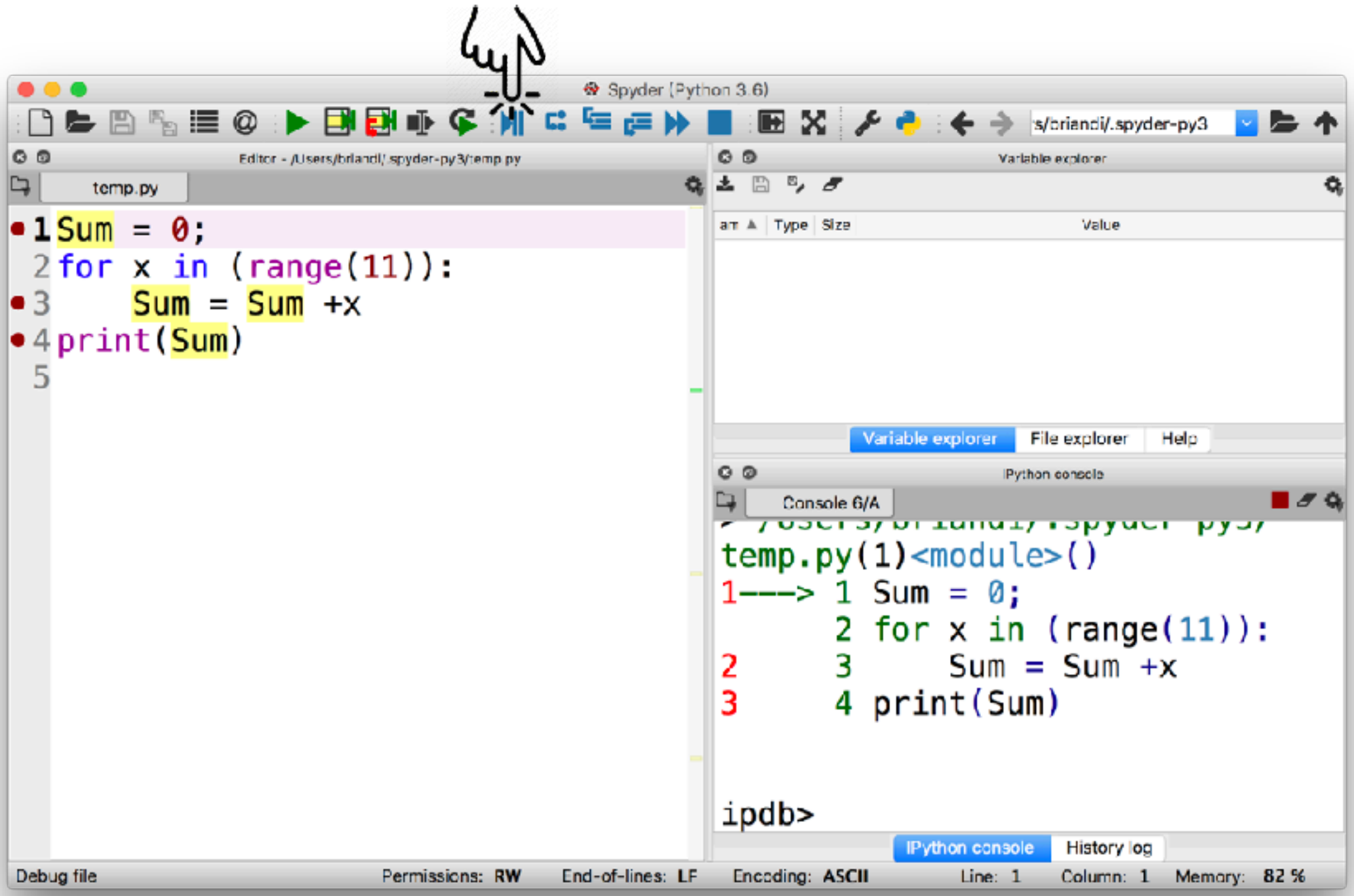
在電腦界中，遇到程式中有錯，就稱之為 bug。 除錯叫做 debug。

Grace Murray Hopper
(1906-1992)

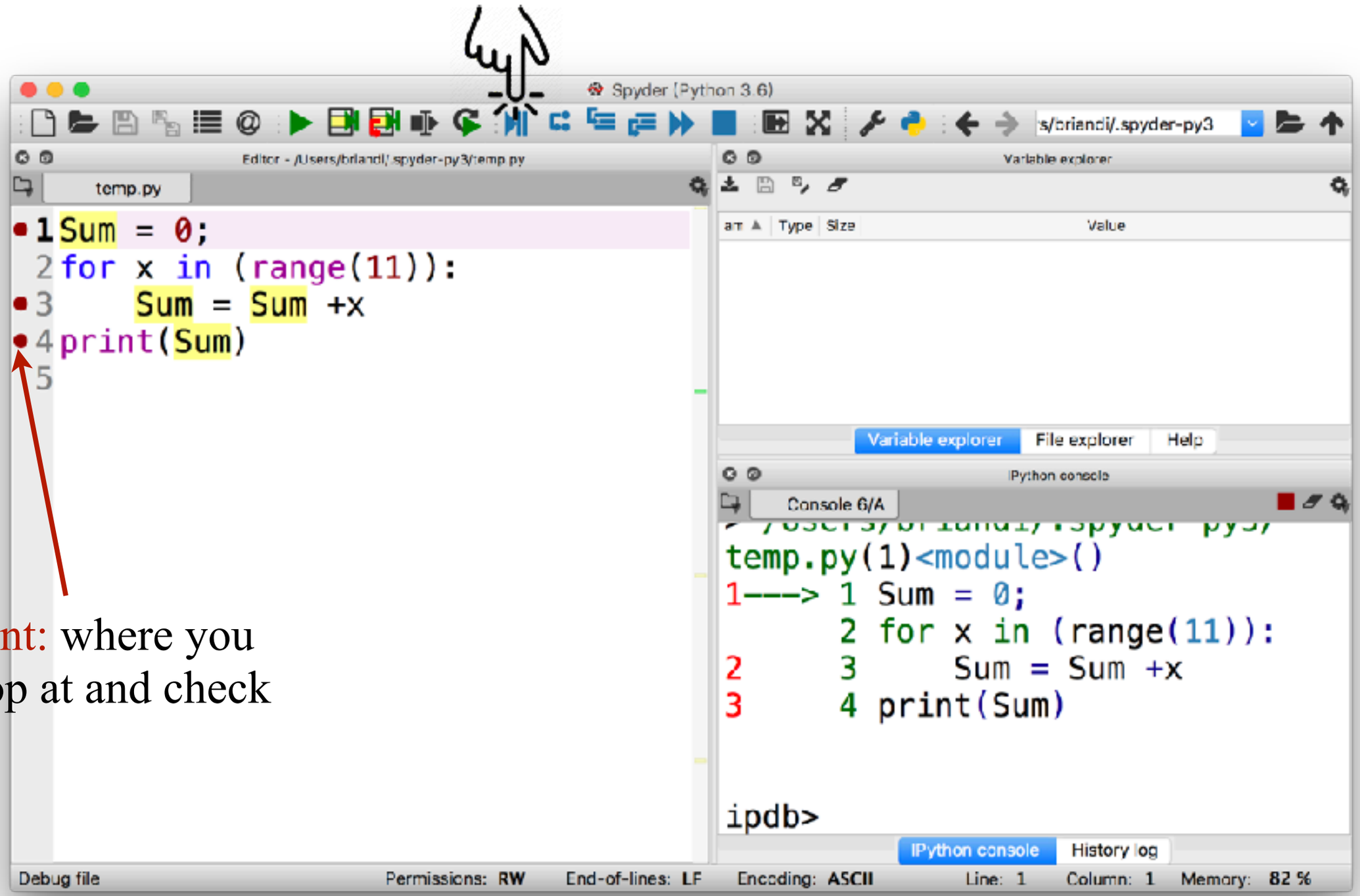
1945 年 9 月 9 日，一隻蛾死在一支繼電器裡面，造成機器當機。經過了一天的檢查，Hopper 找到了那隻蛾， 還把那隻蛾的屍體貼在她的管理日誌上，上面寫著： 「就是這個 bug，害我們今天的工作無法完成。」



Debug in Spyder



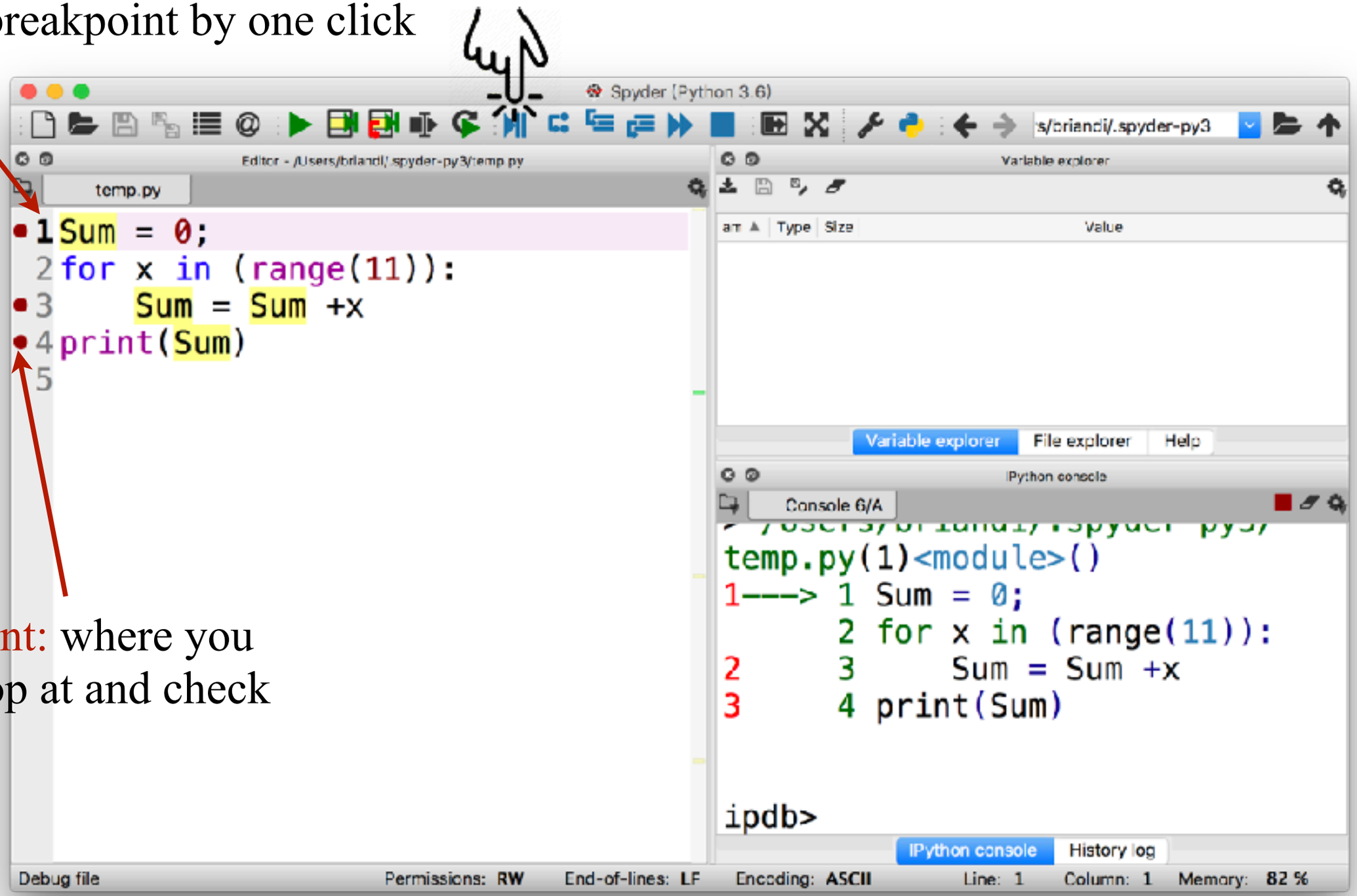
Debug in Spyder



Break point: where you wanna stop at and check values

Debug in Spyder

Line indicator: where you can add or delete breakpoint by one click

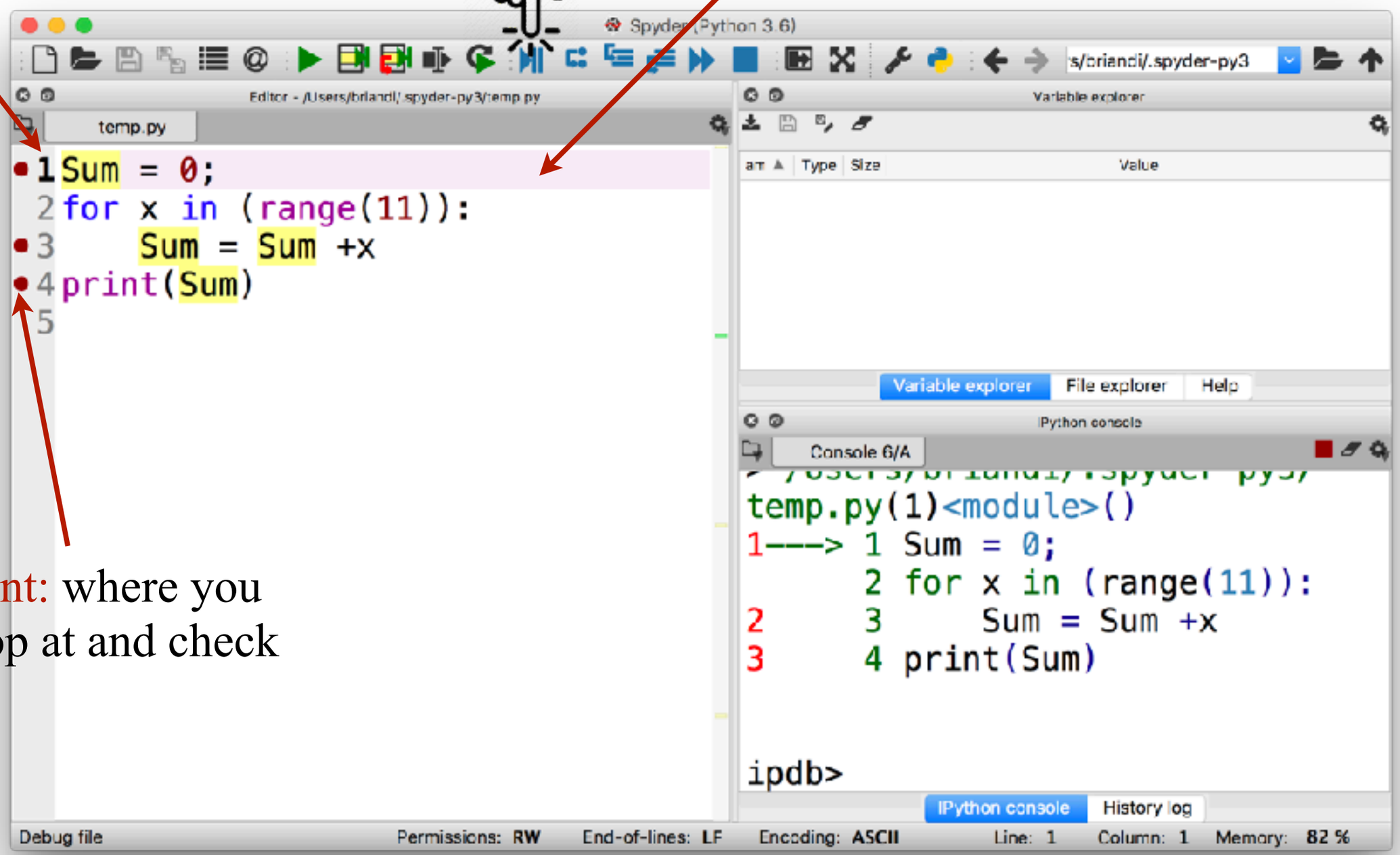


Break point: where you wanna stop at and check values

Debug in Spyder

Line indicator: where you can add or delete breakpoint by one click

The current stage: where you are at now!

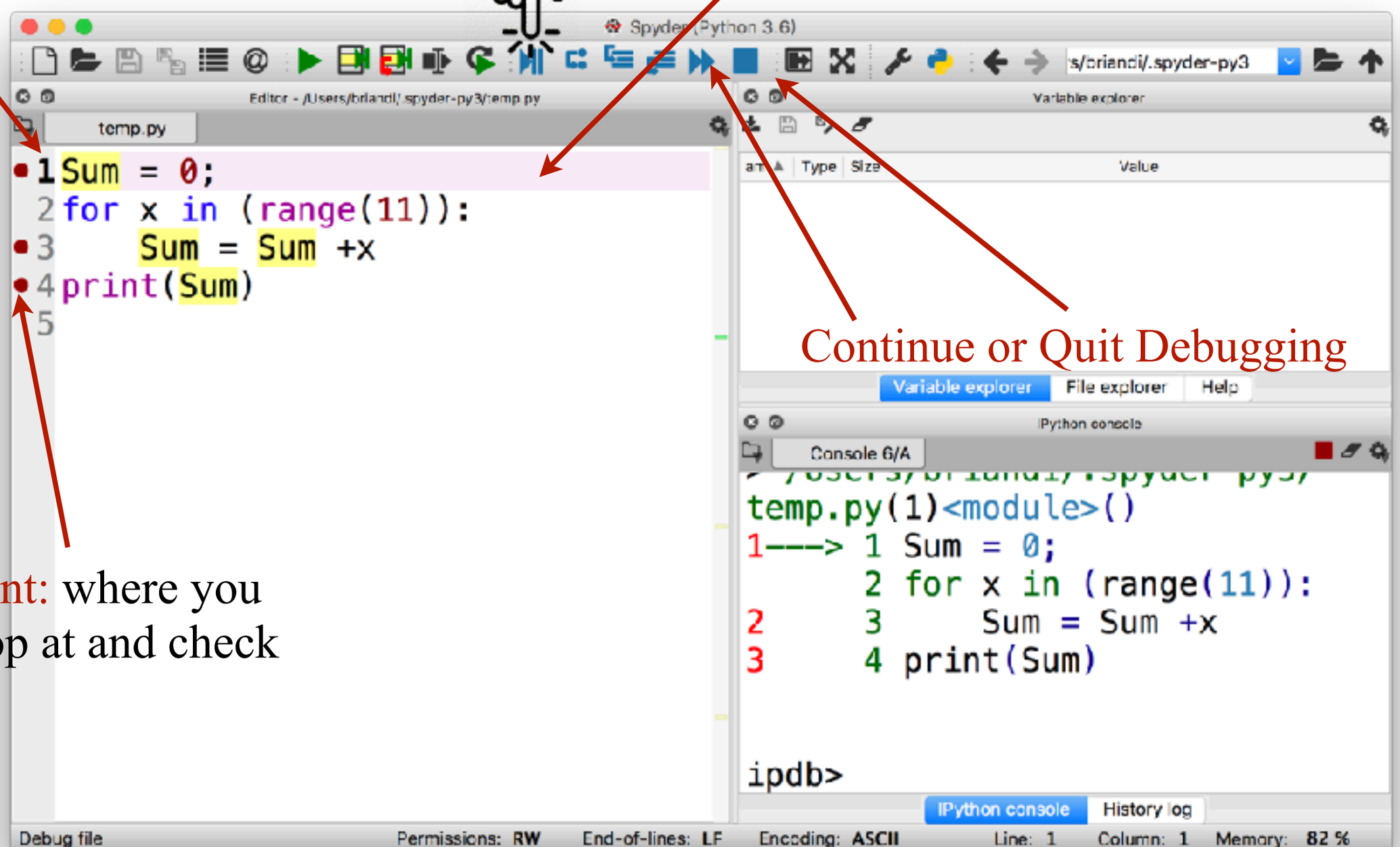


Break point: where you wanna stop at and check values

Debug in Spyder

Line indicator: where you can add or delete breakpoint by one click

The current stage: where you are at now!



Continue or Quit Debugging

Break point: where you wanna stop at and check values

Encapsulating code with functions

- Functions are useful for gathering similar pieces of code in one place.

- Example:

In math

$$x \mapsto f(x) := 2x + 1$$

In Python

```
def f(x):
    return 2*x + 1
```

- Once the function is defined, it can be called using the following code:

```
f(2) # 5
f(1) # 3
```

Anatomy of a function

The start of the function is indicated by the keyword `def`

The function parameters are comma separated.

The function header ends with a colon.

```
def add(arg1, arg2):
    s = arg1 + arg2
    return s
```

Name of the function

The work process of the function

Indentation is **not** optional and shows what block belongs to the function.

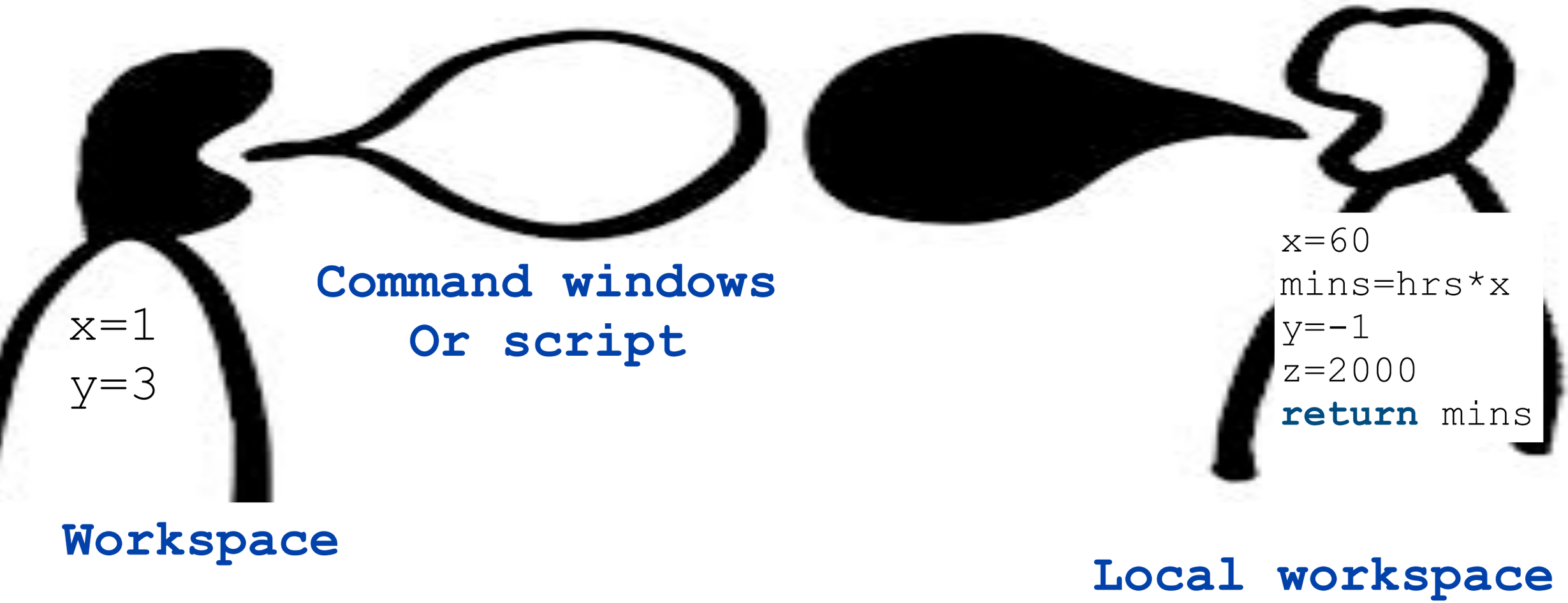
The `return` statement (optional) specif es what is returned by the function. If omitted, the function returns `None`

Important concept of function

- Functions and scripts are similar, but **Function has its local workspace**.
  - Any variables created are available only within that invocation of the function.
  - The variables available to the command line -- those in the base workspace -- are normally NOT visible within the function.
  - In general, the ONLY communication between a function's workspace and that of its caller is through the input and output arguments.

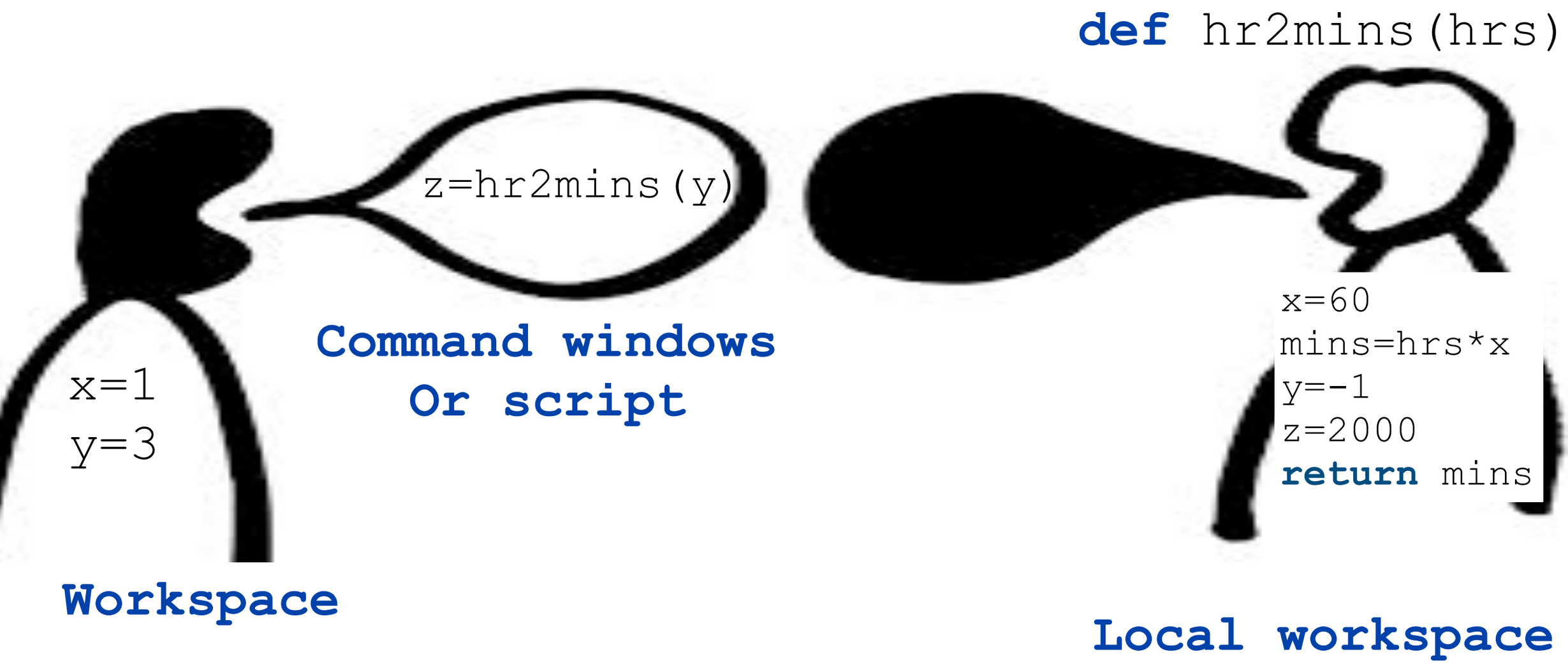
How does a function work?

```
def hr2mins(hrs)
```



Like a good theorem, a good function invites you to inspect the details of its construction once and then **forget** about them.

How does a function work?

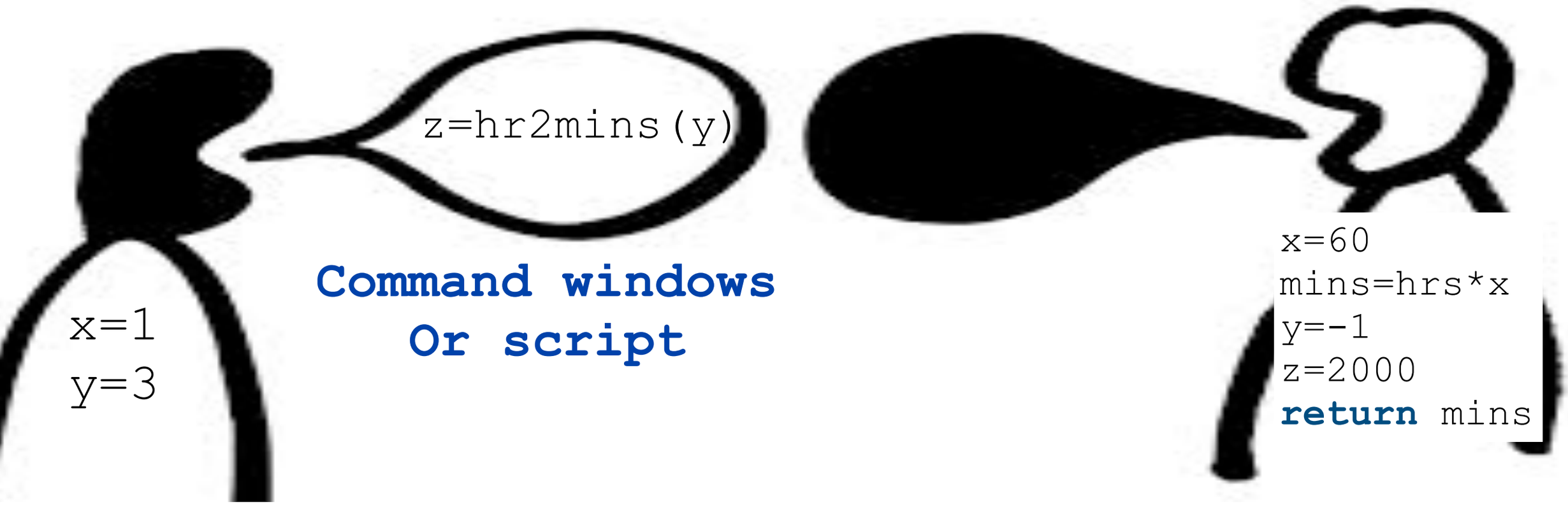


Like a good theorem, a good function invites you to inspect the details of its construction once and then **forget** about them.

How does a function work?

Copy the value of `y` and assign it to `hrs` inside function

```
def hr2mins(hrs)
```

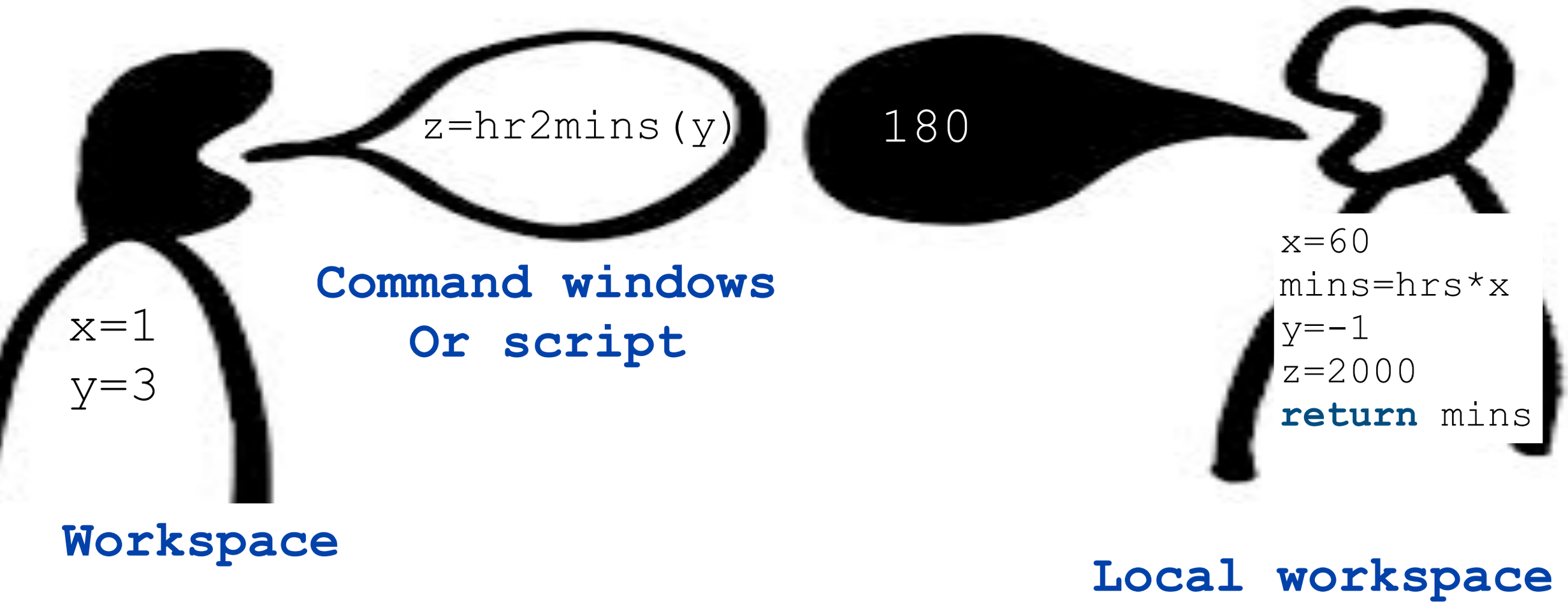


Like a good theorem, a good function invites you to inspect the details of its construction once and then **forget** about them.

How does a function work?

Copy the value of `y` and assign it to `hrs` inside function

```
def hr2mins(hrs)
```

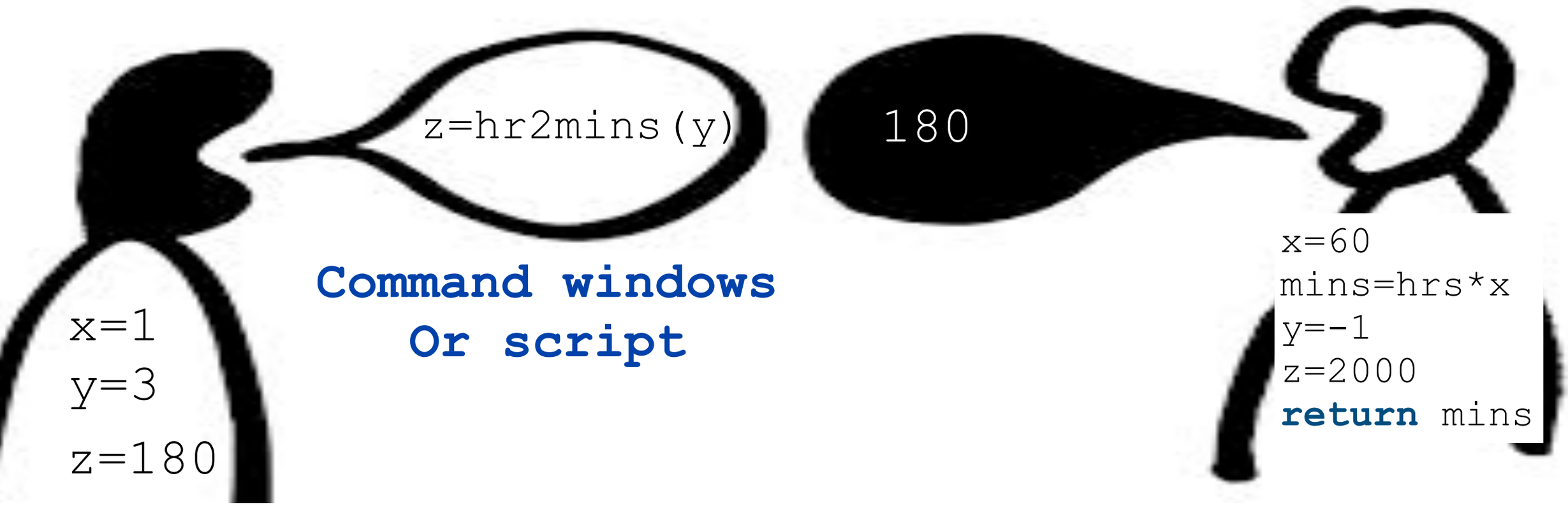


Like a good theorem, a good function invites you to inspect the details of its construction once and then **forget** about them.

How does a function work?

Copy the value of `y` and assign it to `hrs` inside function

```
def hr2mins(hrs)
```



Workspace

Local workspace

Like a good theorem, a good function invites you to inspect the details of its construction once and then **forget** about them.

How does a function work?

STEP1: Prepare the function and save it.
STEP2: Place breakpoints
STEP3: Run it by 

Spyder (Python 3.6)

Editor - /Users/briandi/Dropbox/1_Teaching/201809_Python/Ch2/hr2mins.py

hr2mins.py

```
1  def hr2mins(hrs):
2      x = 60
3      mins = hrs * 60
4      y = -1
5      z = 2000
6      return mins
7
8  x = 1
9  y = 3
10 z = hr2mins(y)
11
12
13
```

Variable explorer

Name	Type	Size	Value
x	int	1	1
y	int	1	3
z	int	1	180

Variable explorer | File explorer | Help

IPython console

Console 1/A

```
In [14]: runfile('/Users/briandi/Dropbox/1_Teaching/201809_Python/Ch2/hr2mins.py', wdir='/Users/briandi/Dropbox/1_Teaching/201809_Python/Ch2')

In [15]:
```

IPython console | History log

Permissions: RW End-of-lines: LF Encoding: UTF-8 Line: 13 Column: 1 Memory: 65 %

Exercise: swap

- Write a function which swap the value of the two positions specified by the user in a list.

```
def swap(data, i, j):
    tmp = data[i]
    data[i] = data[j]
    data[j] = tmp
    return data

L = [1, 2, 3, 4, 6, 5, 7]
swap(L,4,5)
print(L)
```

```
[1, 2, 3, 4, 5, 6, 7]
```

Exercise!

Please write a function to convert NTD to a currency you preferred

and use it!!

Question!

What is the output on the screen if you run this scrip?

```
def f(x):
    print(x)
    return 2*x + 1


x = 3
print(f(2))
print(x)

f(2) # 5
f(1) # 3
```

Question!

What is the output on the screen if you run this scrip?

```
def f(x):
    print(x)
    return 2*x + 1

x = 3
print(f(2))
print(x)

f(2) # 5
f(1) # 3
```

```
2
```

Question!

What is the output on the screen if you run this scrip?

```
def f(x):
    print(x)
    return 2*x + 1


x = 3
print(f(2))
print(x)

f(2) # 5
f(1) # 3
```

```
2
5
```

Question!

What is the output on the screen if you run this scrip?

```
def f(x):
    print(x)
    return 2*x + 1


x = 3
print(f(2))
print(x)

f(2) # 5
f(1) # 3
```

```
2
5
3
```

Question!

What is the output on the screen if you run this scrip?

```
def f(x):
    print(x)
    return 2*x + 1


x = 3
print(f(2))
print(x)

f(2) # 5
f(1) # 3
```

```
2
5
3
2
```

Question!

What is the output on the screen if you run this scrip?

```
def f(x):
    print(x)
    return 2*x + 1


x = 3
print(f(2))
print(x)

f(2) # 5
f(1) # 3
```

```
2
5
3
2
1
```

Quiz!

- Please write a function named `Grade2Letter`
  - Input: grade
  - Output: the corresponding letter A, B, C, D or F

- Please write a script after function `Grade2Letter()` is defined.
  - The script can run `Grade2Letter()` infinite times.
  - In each loop, it will `print 'Enter the grades. Enter a negative number when you finish'`
  - Use `grade = float(input('what is your grade?'))` to get your score, and use `grade` as the input for `Grade2Letter()`
- Hint: you can use `while True:` and `break` to control the loop

Don't copy the ' ' written above, as they may not be the same as those in Spyder.

THE CODE
2018.10.17

E D G E O F T O M O R R O W

IT ONLY TAKES 26 CHANCES (AT LEAST) TO SAVE THE WORLD

WAKE UP
HEATHROW

THE BEACH
SOUTH FRANCE

EXPOSED TO ALIEN

"IS THERE A LOT OF BLOOD?"

TRIES TO SAVE KIMMEL

RAN OVER BY TRUCK

"COME FIND ME WHEN YOU WAKE UP"

ROLLING UNDER TRUCK

TRAINING ARENA
HEATHROW

"I THINK WE BETTER START OVER"